Escuela
Politécnica
Superior

Reconocimiento de emociones humanas y su aplicación a la Robótica Social

Grado en Ingeniería Robótica

Trabajo Fin de Grado

Autor:
Guillermo Gallud Baños
Tutor/es:
Miguel Ángel Cazorla Quevedo
Francisco Rafael Gómez Donoso

Julio 2019

Universitat d'Alacant
Universidad de Alicante

Reconocimiento de emociones humanas y su aplicación a la Robótica Social

Autor
Guillermo Gallud Baños

Tutor/es
Miguel Ángel Cazorla Quevedo
Ciencia de la Computación e Inteligencia Artificial
Francisco Rafael Gómez Donoso
Ciencia de la Computación e Inteligencia Artificial

Grado en Ingeniería Robótica

Escuela
Politécnica
Superior

Universitat d'Alacant
Universidad de Alicante

ALICANTE, Julio 2019

Resumen

En este Trabajo de Fin de Grado se implementa una aplicación que permite al robot semi-humanoide Pepper contar cuentos sociales interactivos. La idea es que el robot sea capaz de reconocer las emociones de la persona que tiene delante y, en función de estas, ir cambiando el transcurso del cuento. Nuestra propuesta tiene como finalidad servir de ayuda en las terapias de niños con autismo y otros trastornos del desarrollo.

Antes de desarrollar la aplicación, se consultan las características y necesidades de los niños con autismo. Seguidamente, se revisan los conceptos básicos de las tecnologías web, y se hace un estudio sobre las posibilidades que brinda el robot Pepper y sobre cómo aplicar el *Deep Learning* a detectar caras y clasificar emociones.

Palabras clave: Robótica Social, Robot Pepper, Deep Learning, Redes Neuronales Convolucionales, Detección de Caras, Clasificación de Emociones.

Agradecimientos

Quisiera dar las gracias a mis tutores, Miguel Cazorla y Francisco Gómez, por su guía y ayuda durante la realización de este trabajo. Asimismo, agradezco al Tribunal su atención.

A mis padres y a mi hermano,
por su apoyo y cariño incondicionales.

Si se espera de una máquina que sea infalible,
es imposible que sea inteligente.

Alan M. Turing

Índice general

Índice de figuras

1 Introducción

Durante los últimos años, se han realizado numerosos estudios sobre la utilidad del uso de robots sociales en la asistencia a personas mayores con demencia, en el cuidado de personas con diversidad funcional, o para facilitar la educación de niños con necesidades especiales, entre otros.

En este trabajo, nos centramos en desarrollar una aplicación destinada a mejorar las habilidades sociales y el aprendizaje de niños con trastorno del espectro autista. Estos niños se caracterizan por tener dificultades a la hora de interactuar socialmente, ya que no son capaces de relacionarse correctamente con otras personas, ni de entender sus sentimientos o intenciones. También experimentan problemas de comunicación, teniendo dificultades con el lenguaje verbal y no verbal, y problemas de entendimiento de gestos y expresiones faciales. Además, demuestran tener una imaginación limitada, lo cual dificulta su desarrollo en el juego. La mayoría de estas personas no pueden vivir de manera independiente una vez alcanzada la edad adulta, por lo que existe la necesidad de intervenciones muy tempranas (Robins y cols., 2004).

Terapias como por ejemplo los cuentos sociales, se han vuelto cada vez más populares para enseñar habilidades de comportamiento apropiadas a los niños con autismo. Estos cuentos son pensados para enseñar a estos niños a cómo jugar mientras se incrementa su habilidad para interactuar con otros niños. Funcionando como un modelo social, el cuento se centra en un personaje en el que el niño pueda verse reflejado y, entonces, se describen los comportamientos, pensamientos y sentimientos del personaje conforme el niño vaya consiguiendo las metas de comportamiento descritas durante la historia (Barry y B. Burlew, 2004).

Por otro lado, las nuevas tecnologías pueden servir de gran ayuda en la mejora de la eficacia de estas terapias. La literatura sugiere que las personas con autismo se sienten más cómodas en entornos predecibles y que prefieren interactuar con objetos, tales como ordenadores o robots, antes que con humanos (Vanderborght y cols., 2012). Un robot, a diferencia de otros juguetes, puede adaptarse individualmente a cada niño y tiene un cierto grado de autonomía que no obliga a la intervención constante de un terapeuta humano (Robins y cols., 2004). Además, robots como NAO o Pepper muestran una cierta apariencia humana, pero proyectan mucha menos información que procesar por el niño, por lo que se reduce el riesgo de sobreestimulación, y se consiguen interacciones más satisfactorias.

Por todo ello, existen estudios como (Vanderborght y cols., 2012) que proponen que sea un robot, y no un terapeuta humano, quien lea los cuentos sociales. Dichos estudios también destacan la necesidad de desarrollar historias interactivas, que permitan al robot responder a las acciones y reacciones del niño.

En este contexto, proponemos una aplicación que permita al robot semihumanoide Pepper contar cuentos sociales que además sean interactivos. Equipamos al robot con la capacidad de reconocer y monitorizar las emociones del niño a lo largo de la historia para que este actúe en consecuencia. El robot también permite al niño tomar la iniciativa y elegir de manera directa qué debe ocurrir en determinados momentos. Mientras tanto, el terapeuta puede ayudar al niño a aprender en función del comportamiento del protagonista de la historia. Esta aplicación también podría ayudar al terapeuta a entender mejor las características especiales del niño, tenga o no autismo.

En definitiva, al igual que en los estudios consultados, nuestro objetivo no es el de sustituir al terapeuta, sino que el robot haga de intermediario entre el niño con autismo y el terapeuta, o incluso entre el niño con autismo y otros niños.

2 Estado del arte y metodología

En este capítulo se van a describir las distintas tecnologías y materiales empleados en el proyecto. Cabe mencionar que a lo largo de este trabajo se han necesitado distintas tecnologías que no son impartidas en el Grado de Ingeniería Robótica, como son las tecnologías web.

2.1 Tecnologías web

En primer lugar, se hace una breve descripción de las distintas tecnologías web que nos permitirán programar una aplicación dinámica para la tablet de Pepper y que el usuario sea capaz de interactuar con ella:

2.1.1 HTML

Hypertext Markup Languaje (HTML) se emplea para la elaboración del contenido de las páginas web. En dicho lenguaje, un elemento externo a la página, como pueden ser una imagen o un vídeo, no se incrusta directamente en el código de la página, sino que se hace una referencia a la ubicación de dicho elemento mediante texto.

Además, HTML es un lenguaje de marcado, por lo que codifica un documento que, junto con el texto antes mencionado, incorpora etiquetas que contienen información adicional acerca de la estructura del texto o su presentación. Entonces, recae en el navegador web la tarea de interpretar el código y unir todos los elementos para visualizar correctamente la página.

2.1.2 CSS

Cascading Style Sheets (CSS) permite dar estilo y diseño a las páginas web. Podemos cambiar el tipo o tamaño de letra de un párrafo, mostrar múltiples columnas o cambiar el color de fondo de la página, entre otras muchas funcionalidades.

2.1.3 JavaScript

JavaScript es un lenguaje de programación interpretado y orientado a objetos, comúnmente empleado en páginas web, cuyas implementaciones permiten al usuario interactuar con dichas páginas. Dicho lenguaje hace posible la manipulación de los elementos del documento HTML respondiendo ante eventos como clicks de botones y otras acciones que el usuario puede ejecutar, ya sea de manera explícita o implícita.

2.2 Robot Pepper

Pepper es un robot semi-humanoide programable que tiene una altura de 120 centímetros. Su diseño, orientado a interactuar con personas, cuenta con una tecnología que le permite analizar el lenguaje verbal y no verbal, la posición de la cabeza y el tono de voz.

Para ello, dispone de una pantalla táctil que le permite recibir y transmitir información, sensores táctiles, micrófonos, cámaras 2D y 3D, y altavoces, entre otros.

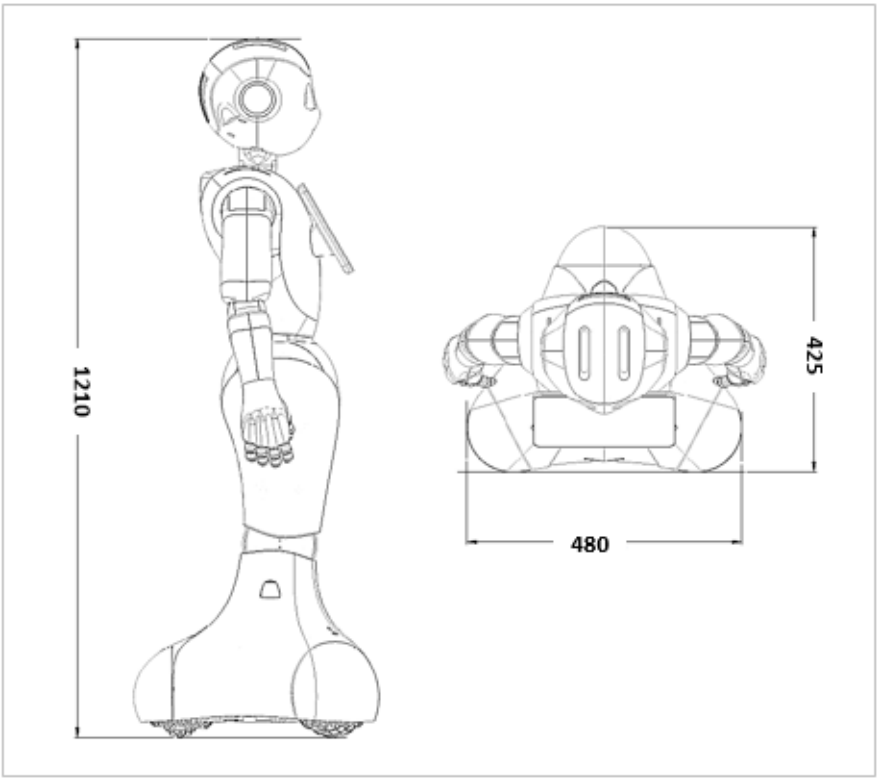


Figura 2.1: Dimensiones

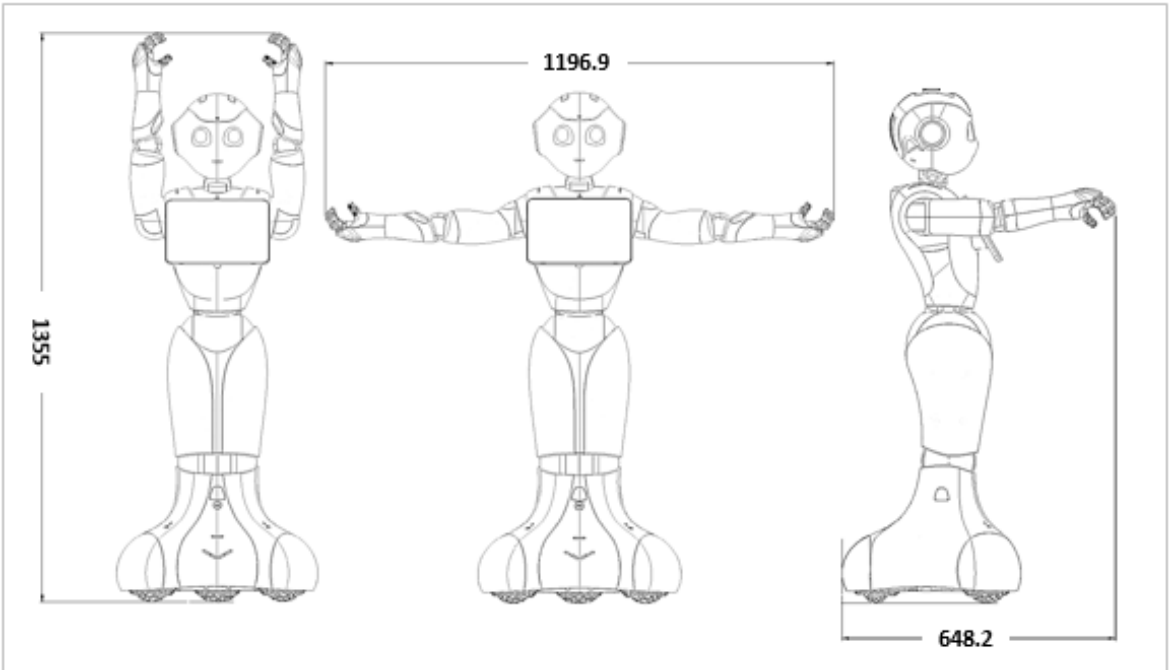


Figura 2.2: Dimensiones con los brazos extendidos

2.2.1 Hardware

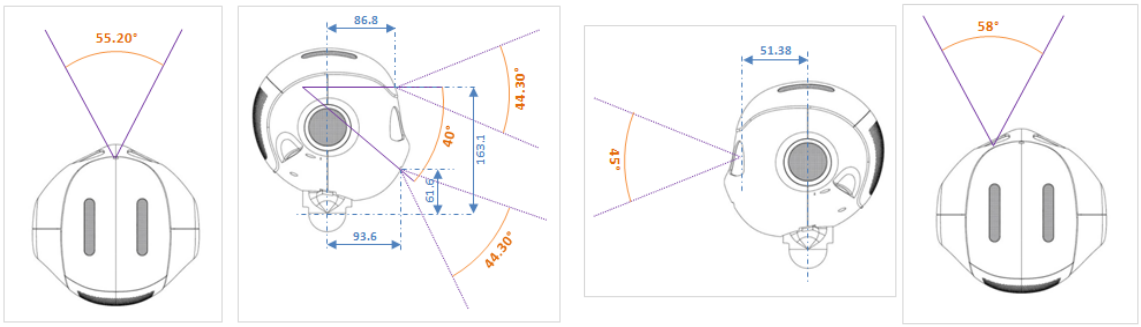
- **Dimensiones**

  El robot Pepper tiene unas dimensiones de (Véase la figura 2.1):

  - 1.210 metros de altura
  - 0.480 metros de largo
  - 0.425 metros de ancho

  Por otro lado, sus dimensiones con los brazos extendidos son las siguientes (Véase la figura 2.2):

  - Hacia arriba: 1.355 metros de altura
  - Hacia los lados: 1.197 metros de largo
  - Hacia delante: 0.648 metros de ancho



(a) Cámaras 2D **(b)** Cámara 3D

Figura 2.3: Cámaras del robot Pepper

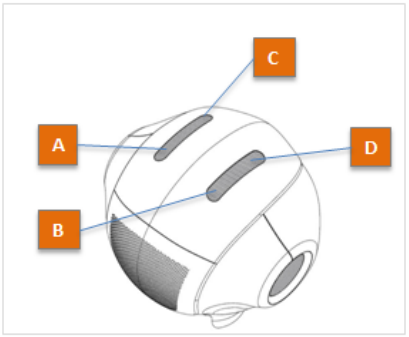


Figura 2.4: Micrófonos

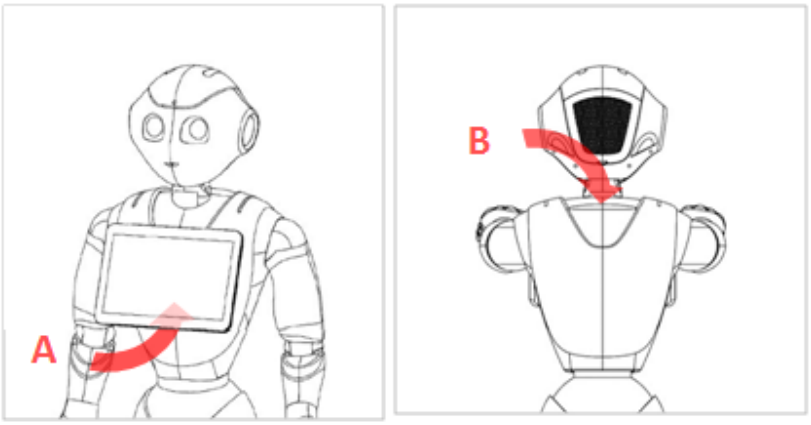


Figura 2.5: Botones

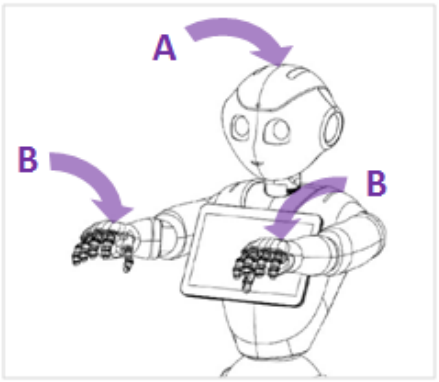


Figura 2.6: Sensores táctiles

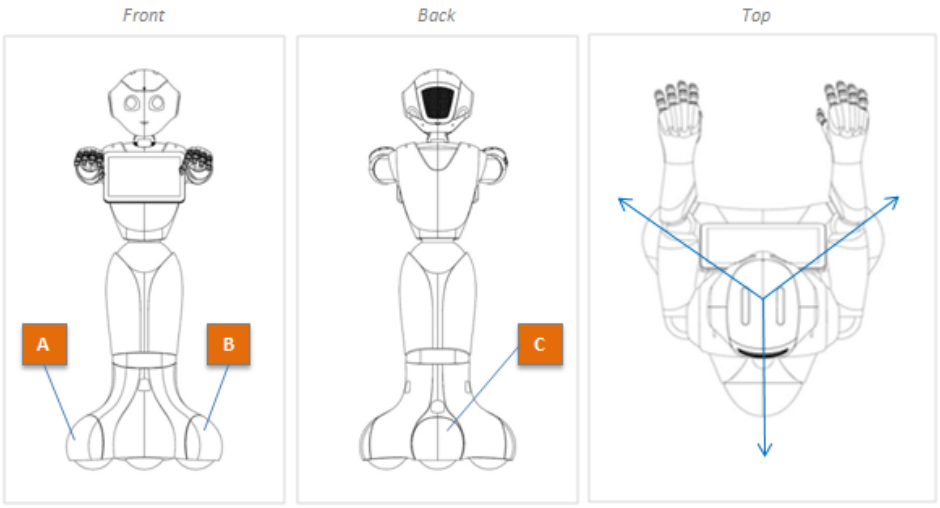


Figura 2.7: Bumpers

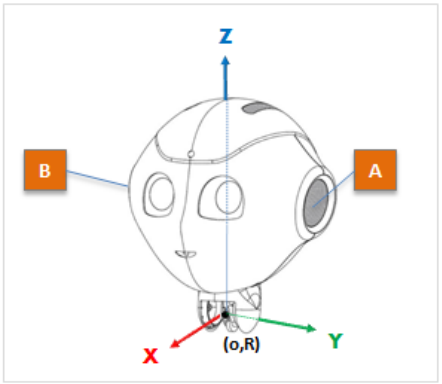
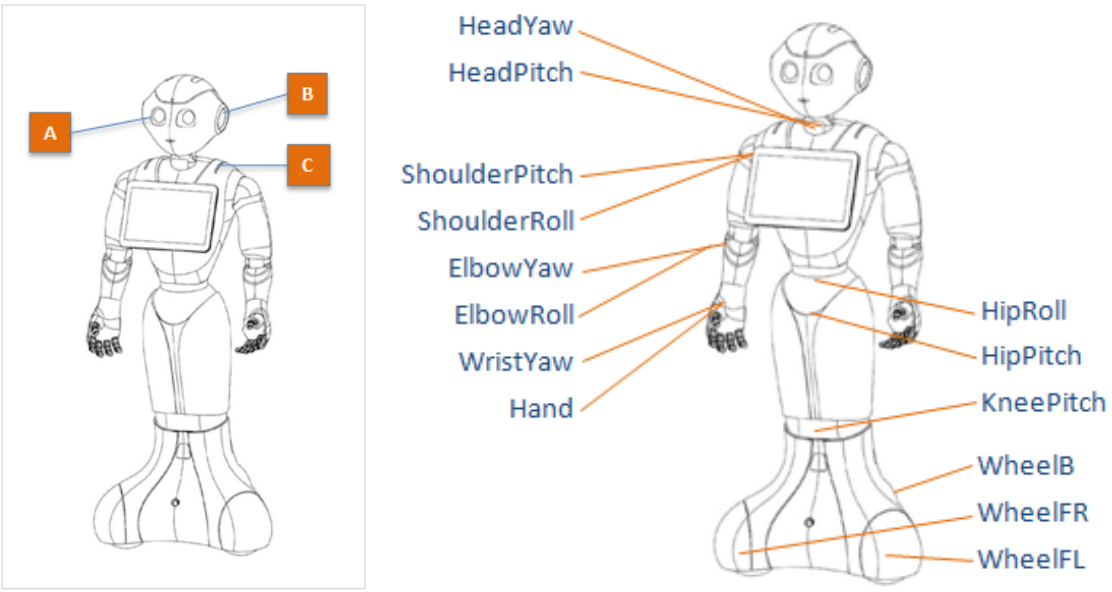


Figura 2.8: Altavoces



(a) Luces LED **(b)** Actuadores

Figura 2.9: Luces LED y actuadores del robot Pepper.

- **Receptores y efectores**

  Receptores. Para poder interactuar de manera eficaz con su entorno, el robot Pepper dispone de una serie de sensores capaces de captar lo que ocurre a su alrededor:

  Cámaras 2D: Dos videocámaras idénticas se localizan en la cabeza de Pepper, una en la frente y la otra en la boca. Proporcionan una resolución de hasta 2560x1920 con una frecuencia de 1 fps. A 30 fps, alcanza una resolución de 640x480. Véase la Figura 2.3a.

  Cámara 3D: Un sensor ASUS Xtion 3D se encuentra tras los ojos del robot. Alcanza una resolución de 320x240 con una frecuencia de 20 fps. Véase la Figura 2.3b.

  Micrófonos: Disponemos de cuatro micrófonos sobre la cabeza del robot. Véase Figura 2.4.

  Sensores táctiles y de contacto Contamos con dos botones, uno en el pecho y otro entre los hombros (Figura 2.5). El primero serviría para encender y apagar el robot de manera normal mientras que el de la espalda sería el botón de parada de emergencia. También disponemos de tres sensores táctiles, uno en la cabeza y uno en cada mano (Figura 2.6). Por último, Pepper también tiene tres bumpers (Figura 2.7).

Otros sensores: Además de los anteriores, el robot Pepper cuenta con un giroscopio y un acelerómetro, 6 láseres de línea, 2 sensores infrarrojos, 2 sónares y 30 encoders magnéticos rotativos.

Efectores. Emiten unas determinadas respuestas tras haber sido procesada la información recibida por los receptores:

Altavoces: Situados en las orejas de Pepper. Véase la figura 2.8.

Luces LED: Localizados en los ojos, en las orejas y en los hombros. Véase la figura 2.9a.

Actuadores: Contamos con 20 grados de libertad para conseguir movimientos más naturales y expresivos. Los actuadores del robot se encuentran repartidos por todo su cuerpo, tal y como puede observarse en la figura 2.9b.

Tablet: La cual se encuentra en el pecho del robot. Dicho dispositivo actúa a la vez de receptor y de efector, ya que no solo mostrará el texto, las imágenes y los vídeos que deseemos, sino que también podremos interactuar con ella mediante el tacto. Debido a su versatilidad, será uno de los elementos que más usemos. No obstante, recordemos que nuestra aplicación debe ser accesible para personas con diversidad funcional, por lo que no puede ser el único medio de interacción con el niño o la niña.

2.2.2 Software

En cuanto a la programación, nos basamos en NAOqi, que es una distribución de GNU/Linux desarrollada específicamente para Pepper y los demás robots de Aldebaran.

Podemos hacer uso tanto del lenguaje Python como de C++. Además, existe la posibilidad de utilizar Choreographe, la cual es una aplicación de escritorio multiplataforma que proporciona una programación más gráfica y sencilla.

No obstante, para nuestro propósito, nos decantamos por Python debido a que permite un prototipado mucho más rápido. Además, Keras, librería de Deep Learning que usamos, solo se encuentra disponible para dicho lenguaje. Por otro lado, no se hace uso de Choreographe debido a que el estilo de diseño basado en eventos y funciones no tiene expresividad suficiente para implementar el proyecto que se propone.

Retomando NAOqi, es un framework de programación que da respuesta a las necesidades comunes de un robot, como son el paralelismo, un funcionamiento síncrono o el manejo de eventos. Dicho framework permite una comunicación homogénea entre diversos módulos (movimiento, audio, vídeo), distintos lenguajes de programación, así como un uso compartido de la información.

- **Características principales**

  NAOqi es multiplataforma y multilenguaje, lo que permite la creación de aplicaciones distribuidas. Veamos qué significan estos conceptos:

  Multiplataforma: NAOqi puede ser utilizado en Windows, Linux y en MacOS. Si usamos C++, al ser un lenguaje compilado, tendremos que compilar nuestro código específicamente para el sistema operativo de destino. Por este motivo, para ejecutar código C++ en el robot, primero tendremos que usar una herramienta de compilación cruzada para generar el código máquina entendible por el sistema operativo del robot: NAOqi OS.

  En cambio, como Python es un lenguaje interpretado, no será necesario compilar el código, por lo que será mucho más cómodo de ejecutar tanto en el ordenador como directamente en el robot. Esta es una de las razones por las que decidimos trabajar en Python.

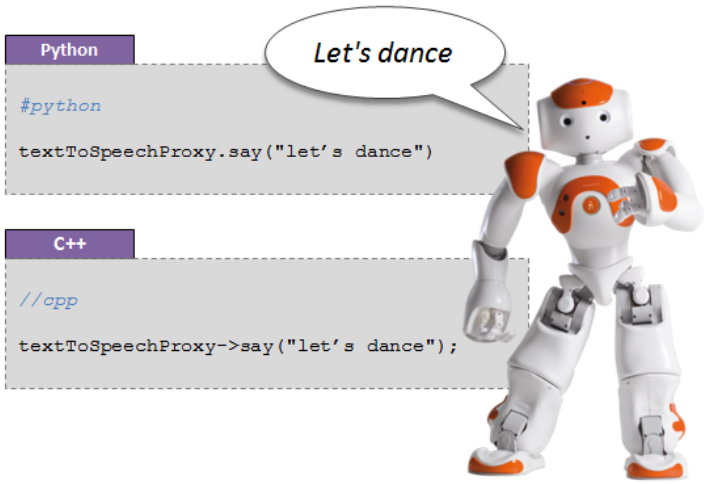


Figura 2.10: Métodos de programación en Python y C++.

  Multilenguaje: Como ya hemos comentado, el software del robot puede ser desarrollado tanto en Python como en C++. En ambos casos, los métodos de programación son exactamente los mismos. Véase la figura 2.10.

  Aplicaciones distribuidas: Una aplicación en tiempo real puede ser no solo un ejecutable, sino varios procesos y/o módulos distribuidos en distintas máquinas. En cualquier caso, los métodos son siempre los mismos. Como veremos más adelante, esta característica nos permitirá dividir nuestra aplicación entre el robot Pepper, un PC y un servidor con potentes tarjetas gráficas.

- **El proceso NAOqi**

  Un ejecutable NAOqi que corre en el robot es lo que conocemos por **broker**. Cuando este empieza a funcionar, carga un fichero con preferencias llamado autoload.ini, que define qué bibliotecas deben cargarse. Cada una de estas bibliotecas contiene uno o más módulos que el broker usará para publicar sus métodos. Véase la figura 2.11.

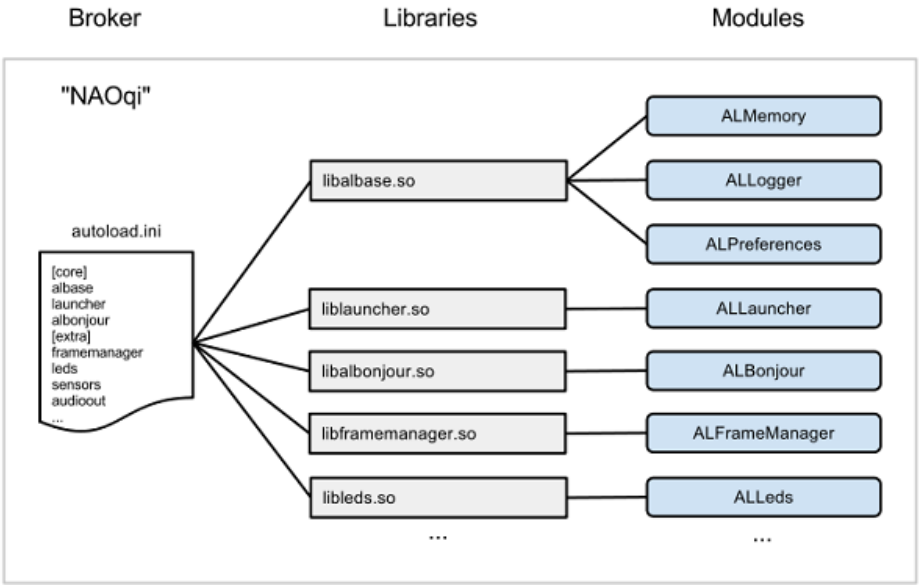


Figura 2.11: Fichero autoload.ini y las bibliotecas a las que se llama.

El broker proporciona servicios de búsqueda para que los módulos de la red puedan encontrar cualquiera de los métodos que hayan sido publicados.

Al cargar algún módulo, formamos un árbol de métodos asociados a módulos, y de módulos asociados a un broker. Véase la figura 2.12.

Antes de seguir, profundicemos en los conceptos de broker, módulo y método:

Broker: Es un objeto que proporciona:

- **Servicios de directorio:** Permitiéndonos encontrar módulos y métodos.
- **Acceso a la red:** Haciendo posible llamar desde fuera del proceso a los métodos pertenecientes a los módulos que hayan sido declarados.

Gracias a esto, podríamos, por ejemplo, llamar desde nuestro ordenador a *setIntensity()*, perteneciente a *ALLeds*, y manipular los LEDs del robot Pepper. También podría un Pepper acceder al broker de otro robot y hacerle hablar.

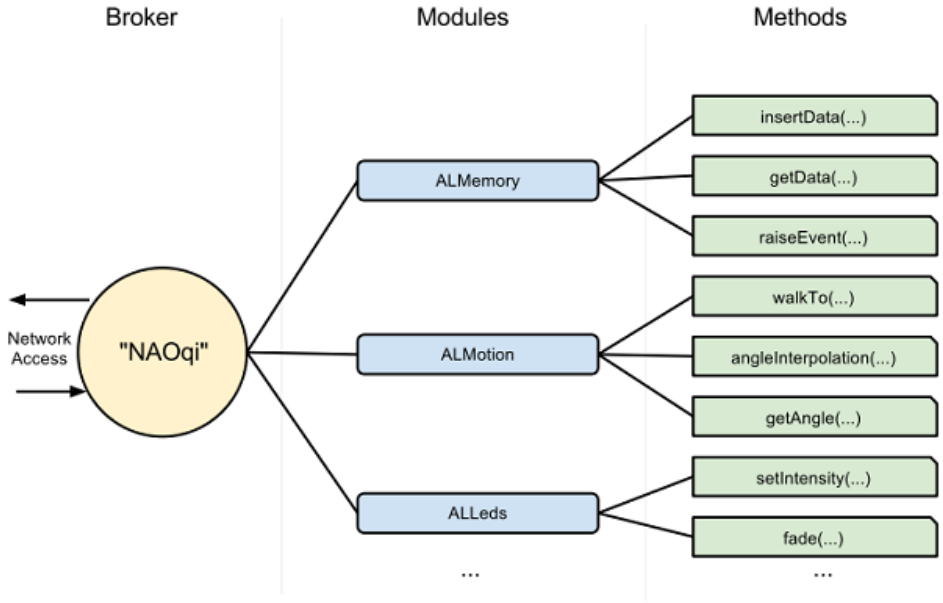


Figura 2.12: El árbol formado por un broker, sus módulos y sus métodos.

En cualquier caso, casi nunca tendremos que preocuparnos por los brokers. Estos harán su trabajo de manera transparente, permitiéndonos programar de la misma forma tanto para llamadas a módulos locales (en el mismo proceso) como para llamadas a módulos remotos (en otro proceso u otra máquina).

Proxy: Es un objeto que representa un módulo y se comportará como este. Por ejemplo, si creamos un proxy al módulo *ALMotion*, obtendremos un objeto que contendrá los métodos de dicho módulo.

Para crear un proxy a un módulo, y por tanto llamar a sus métodos, tenemos dos opciones:

- Podemos simplemente usar el nombre del módulo: En este caso, el código que estemos ejecutando y el módulo al cual queramos acceder deberán encontrarse en el mismo broker. Esto se conoce como llamada local.

- También podemos utilizar, además del nombre del módulo, la IP y el puerto del broker: En este caso, el módulo llamado deberá estar en el correspondiente broker. Esta situación será la que más nos interese, puesto que el código principal de nuestra aplicación se ejecutará en un servidor desde el cual querremos acceder a los métodos del robot Pepper para hacerle hablar, mostrar información a través de su tablet, para obtener las imágenes que tomen sus cámaras, etc.

Módulos: Normalmente, un módulo es una clase dentro de una biblioteca. Al cargar una biblioteca especificada en el fichero *autoload.ini*, dicha clase será automáticamente instanciada.

Un módulo puede ser tanto local como remoto:

- Los **módulos locales** son dos o más módulos que son lanzados en el mismo proceso. Se comunican entre ellos mediante un solo broker. Pueden compartir variables y llamar a los métodos del otro de manera directa, por lo que la comunicación entre ellos es muy rápida y eficiente.

- Por el contrario, los **módulos remotos** se comunican a través de la red. Un módulo necesita un broker para comunicarse con otros módulos. A continuación, nos centraremos en este tipo de módulos.

Comunicación entre módulos remotos: Esta puede darse de dos maneras:

- Mediante una conexión **broker-a-broker**: Se da una comunicación recíproca y ambos módulos pueden acceder a los métodos del otro.

- Mediante una conexión **proxy-a-broker**: El proxy podrá acceder a todos los módulos suscritos al broker, pero los módulos suscritos al broker en ningún caso podrán acceder al módulo representado por el proxy. Es decir, la comunicación se dará en un solo sentido. En nuestro caso, como tan solo querremos acceder a los métodos del robot desde el servidor, usaremos este tipo de conexión.

Llamadas no bloqueantes: Aparte de las bloqueantes, simples funciones que deben terminar de ser ejecutadas antes de que sean llamadas las siguientes, contamos con las llamadas no bloqueantes. Para ello, usaremos el objeto *post* de un proxy, el cual creará una tarea que se ejecutará en un hilo paralelo. Véase la figura 2.13. Esto permitirá que nuestro Pepper hable y mueva la cabeza simultáneamente, p.ej.

Memoria del robot: *ALMemory* es la memoria del robot. Todos los módulos pueden leer o escribir datos, suscribirse a eventos, así como ser llamados cuando un eventos se produzca. *ALMemory* es un array de lo que se conoce como *ALValues*. *ALMemory* contiene tres tipos de datos:

- Datos de los efectores y los sensores.

- Eventos.

- Micro-eventos, más rápidos que los eventos.

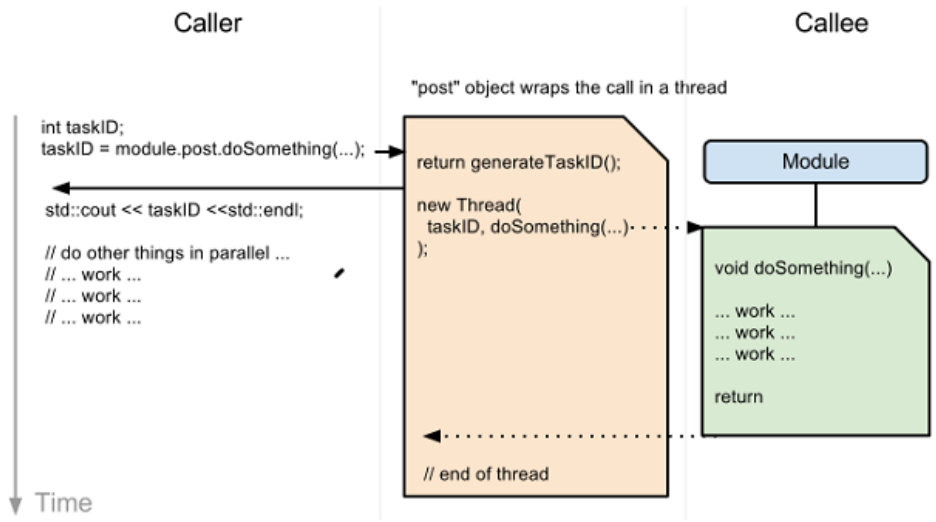


Figura 2.13: Llamada a un método no bloqueante.

Manejo de eventos: Determinados módulos publican lo que llamamos eventos. Para suscribirnos al evento de un módulo, debemos usar un callback que sea un método de nuestro suscriptor. Por ejemplo, podemos usar un módulo llamado *FaceReaction*, el cual contiene el método *onFaceDetected*. Contemos también con el módulo *ALFaceRecognition*, que contiene el método *FaceDetected*. Entonces, podemos suscribir el módulo *FaceReaction* al método *FaceDetected* (de *ALFaceRecognition*) mediante la función callback *onFaceDetected*. En ese caso, cada vez que se detecte una cara, se lanzará un evento que avisará de que hay una cara frente a la que reaccionar. En nuestra aplicación usaremos este mismo enfoque para comunicar el código Python que controla el robot con el código JavaScript que se ejecuta en la tablet de Pepper.

2.3 Fundamentos del Machine Learning

Antes de centrarnos en las Redes Neuronales Profundas, convendría repasar una serie de conceptos básicos:

2.3.1 Inteligencia Artificial

La Inteligencia Artificial (IA) sería el término más general (véase la figura 2.14) y engloba al campo del Aprendizaje Automático (Machine Learning en inglés), pero también otras técnicas como los algoritmos de búsqueda, el razonamiento simbólico, el razonamiento lógico y la estadística. A su vez, el Machine Learning engloba el Aprendizaje Profundo (Deep Learning).

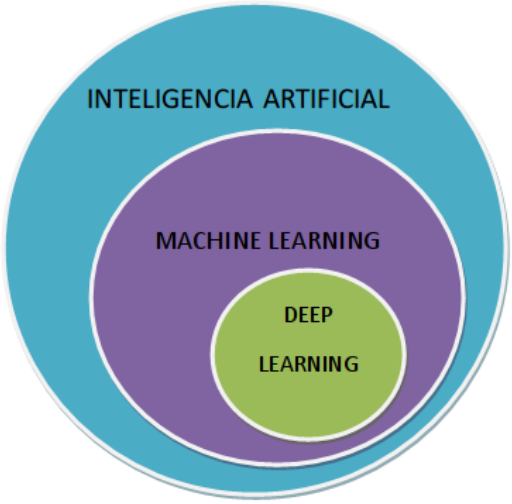


Figura 2.14: Diagrama de la inteligencia artificial.

Aunque la IA sigue siendo un concepto ambiguo, una posible definición sería: el esfuerzo por automatizar las tareas intelectuales que normalmente realizan los seres humanos.

2.3.2 Machine Learning

El Machine Learning se refiere a un amplio conjunto de técnicas informáticas que nos permiten dar a una máquina la capacidad de aprender sin ser explícitamente programada. Más en concreto, el Machine Learning consiste en crear programas capaces de generalizar comportamientos a partir de información suministrada en forma de ejemplos. Es, por lo tanto, un proceso de inducción del conocimiento.

Dentro de este campo encontramos una gran cantidad de algoritmos, entre los que se destacan el aprendizaje por refuerzo, los árboles de decisión o el Aprendizaje Profundo (Deep Learning en inglés).

Dichos algoritmos pueden ser clasificados en dos tipos:

De aprendizaje supervisado: En este caso, tenemos una serie de variables de entrada y una determinada función que transforma dichas entradas en variables de salida:

$$y = f(x)$$

Para obtener esta función, se debe entrenar el sistema con un conjunto de datos de entrada etiquetados. Conociendo las entradas (las características de estas) y también sus correspondientes salidas (las etiquetas), se van corrigiendo los parámetros de la

función en un proceso iterativo. Tras este entrenamiento, la función de transformación será capaz de predecir con una determinada precisión las etiquetas de nuevos datos de entrada cuyos valores de salida sean desconocidos.

A su vez, los problemas de aprendizaje supervisado se pueden dividir en:

Clasificación: Cuando la variable de salida sea una categoría, un valor cualitativo. Por ejemplo, tumor maligno/benigno en función del tamaño del tumor, la forma, etcétera.

Regresión: Cuando la salida sea un valor cuantitativo: el precio de una determinada casa en euros en función de la antigüedad, el barrio en el que se encuentre, etc.

El reconocimiento de emociones que emplea nuestra aplicación se basa en aprendizaje supervisado de clasificación. En nuestro caso, la máquina recibe como variables de entrada las imágenes que toma la cámara del robot. Entonces, una función (o mejor dicho una red de funciones, como ya veremos) predice si, de haberla, la persona de la imagen está contenta, triste, etcétera.

De aprendizaje no supervisado: En este caso los datos de entrada no tienen etiquetas. El objetivo de este tipo de algoritmos es el estudio de la estructura o distribución del conjunto de datos.

2.3.3 Deep Learning: Redes Neuronales Profundas

Llamamos Deep Learning al conjunto de algoritmos que intenta modelar abstracciones de alto nivel haciendo uso de arquitecturas compuestas de transformaciones no lineales múltiples, es decir, haciendo uso de Redes Neuronales Profundas. Para problemas sencillos de clasificación, podemos emplear regresión logística simple añadiendo características cuadradas o cúbicas, por ejemplo. No obstante cuando el número de características es muy elevado, para conseguir una precisión aceptable necesitaríamos añadir un número desorbitado de características, por lo que el coste computacional sería inasumible.

Es por ello que para problemas más complejos, como es el reconocimiento de imágenes, se utilizan las Redes Neuronales Profundas.

Una Red Neuronal Profunda es un modelo computacional vagamente inspirado en el comportamiento de su homólogo biológico. Consiste en una red como la mostrada en la figura 2.15, donde contamos con una capa de entrada, múltiples capas ocultas y una última de salida. Cada nodo en cada capa puede estar conectado a todos o solo a algunos de los nodos de su siguiente capa. Cuantas más capas ocultas añadamos, más profunda será la red. Además, dichas capas forman una jerarquía de características desde un nivel de abstracción más bajo a uno más alto.

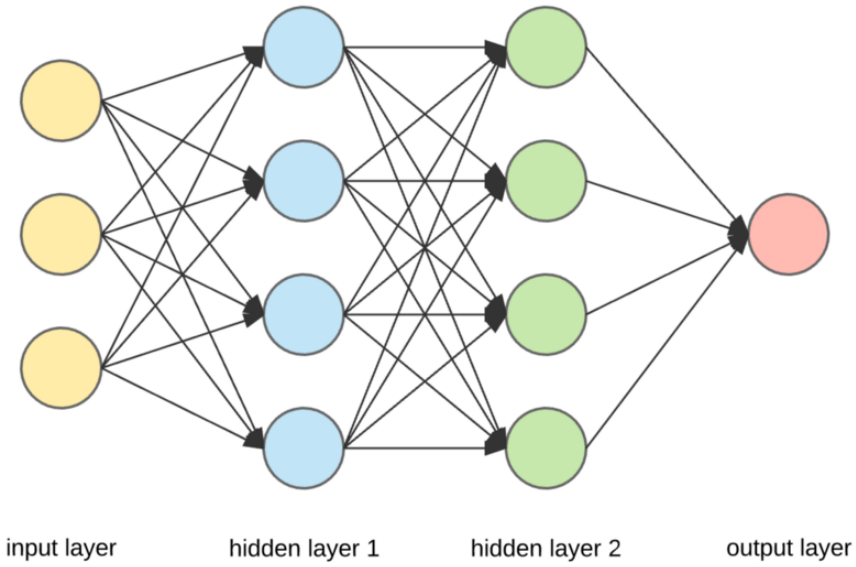
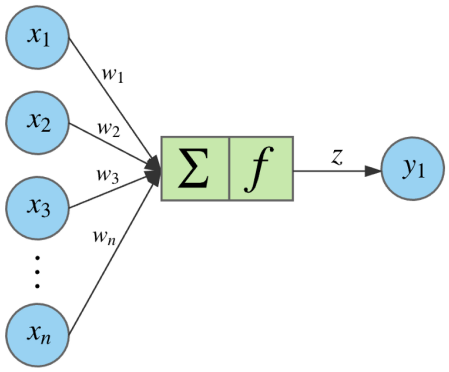


Figura 2.15: Red neuronal artificial.



(a) Neurona artificial

$$z = f(b + x \cdot w) = f\left(b + \sum_{i=1}^{n} x_i w_i\right)$$

$$x \in d_{1\times n},\ w \in d_{n\times 1},\ b \in d_{1\times 1},\ z \in d_{1\times 1}$$

(b) Ecuación de una neurona. x es el vector de entradas, w es el vector de pesos, b es el término bias y z es la salida de la neurona.

Figura 2.16: Neurona artificial y su ecuación.

En cada uno de los nodos de las capas ocultas encontraremos lo que se muestra en la figura 2.16a. Dichos nodos reciben el nombre de neuronas artificiales. Cada neurona toma la suma ponderada de sus entradas, multiplicadas por pesos, y la procesa mediante una función de activación no lineal como puede ser la función sigmoide. Esta señal será la salida de la neurona en cuestión, que a su vez será una de las entradas a las neuronas de la capa siguiente. Por tanto, las señales se transmiten de izquierda a derecha hasta llegar a la última capa, en lo que conocemos como propagación hacia adelante. La primera capa constituye las características de entrada. En la última capa, cada neurona será una posible salida. Si la red está bien entrenada, la neurona de salida que más se active se corresponderá con la respuesta correcta.

Llegado este punto, probablemente nos estemos preguntando cómo se entrena una red. Al igual que antes, tendremos un conjunto de datos etiquetados con los que ajustaremos los

parámetros de las funciones de cada neurona de manera iterativa hasta conseguir que la red se comporte como deseamos. Los parámetros de dichas neuronas son lo que antes hemos llamado pesos y el proceso en que se ajustan dichos pesos se conoce como propagación hacia atrás, durante la cual la señal se transmite de derecha a izquierda. Durante este proceso, se hace uso de la llamada función de coste.

Recapitulando, tenemos dos procesos: propagación hacia adelante (forward propagation) y propagación hacia atrás (backpropagation).

Una vez la red esté correctamente entrenada, solo se ejecutará el forward propagation, recibiendo unas determinadas características de entrada y calculando una salida adecuada en función de los pesos correctamente ajustados previamente.

No obstante, en la fase de entrenamiento, se ejecutarán ambos procesos. A continuación, se explica dicha fase de entrenamiento y se profundiza en algunos de los conceptos antes expuestos:

1. **Inicialización de pesos aleatorios.** Dichos valores se encontrarán en un rango de 0 a 1.

2. **Forward propagation.** Las señales serán transmitidas desde la capa de entrada hasta la de salida, pasando por las capas ocultas, en las cuales las funciones de activación procesan los productos de las sumas ponderadas de las entradas a las neuronas por sus respectivos pesos (Véase la figura 2.16b). Estas funciones imponen un umbral que se debe sobrepasar para que una señal se transmita a otra neurona. A su vez, los pesos pueden incrementar o inhibir el estado de activación de las neuronas adyacentes. De este modo, tras llegar a la última capa, obtenemos una determinada salida, la hipótesis.

3. **Cálculo de la función de coste.** Dicha función medirá el error de nuestra red. Las salidas obtenidas en el paso anterior serán comparadas con las salidas esperadas, es decir, las etiquetas conocidas.

4. **Backpropagation.** Desde la capa de salida hasta la de entrada, propagamos el error a cada neurona. Es decir, calculamos la contribución de cada peso al error global. En función de esto, ajustamos los pesos mediante el descenso por gradiente u otras técnicas de optimización.

5. **Iteramos** desde el paso 2 hasta el 4 con todo el conjunto de entrenamiento hasta conseguir una precisión aceptable.

Una vez introducidos los aspectos y el funcionamiento básicos de una red neuronal, hablemos de una variación llamada Red Neuronal Convolucional, la cual está adaptada a la visión artificial, ideal, por tanto, para que nuestro robot reconozca emociones.

2.4 Clasificación de objetos: Redes Neuronales Convolucionales

Al igual que en las redes neuronales ordinarias, tenemos una capa de entrada, al menos otra oculta y una última de salida. La diferencia principal es que esta vez, las neuronas de las capas están distribuidas en tres dimensiones: altura, anchura y profundidad, correspondiéndose esta última con los canales de color RGB. Dichas matrices reciben el nombre de filtros.

Una Red Neuronal Convolucional (CNN) es un algoritmo de Deep Learning que recibe como entrada una imagen, asocia una determinada importancia (pesos y umbrales entrenables) a los distintos aspectos u objetos que aparecen en ella y es capaz de diferenciar dichos objetos unos de otros. El preprocesado de las imágenes es mucho menor del requerido por otros algoritmos de clasificación. Por otro lado, mientras que en los primeros métodos, los filtros se diseñaban a mano, las CNN son capaces de aprender qué filtros son mejores para cada aplicación.

Antes de seguir, puede que se nos ocurra lo siguiente: una imagen no es más que una matriz numérica de píxeles, ¿por qué no simplemente transformar dicha matriz en un vector lineal e introducirlo directamente en una red neuronal ordinaria? Pues debido a que los píxeles de un determinado objeto dentro de una imagen no están relacionados de manera directa con aquellos píxeles que se encuentren lejos de este. Por ejemplo, para reconocer una cara sonriente, no nos importa el brazo derecho de la persona, o el lugar donde se encuentre, sino aquellos píxeles correspondientes a la boca, los ojos, las líneas de expresión, etc. Es esa información estructural contenida en regiones locales de la imagen, esas dependencias espaciales, lo que motiva el uso de filtros en vez de neuronas simples.

Mediante el uso de filtros, solo aquella zona de la imagen cubierta en el momento por dicho filtro estaría conectada a las neuronas de la capa siguiente. Al reducir el número de conexiones, el número de pesos también baja y, por ello, el coste computacional será mucho menor. A su vez, al evitar los píxeles lejanos al objeto, reducimos el overfitting y la red generaliza mejor.

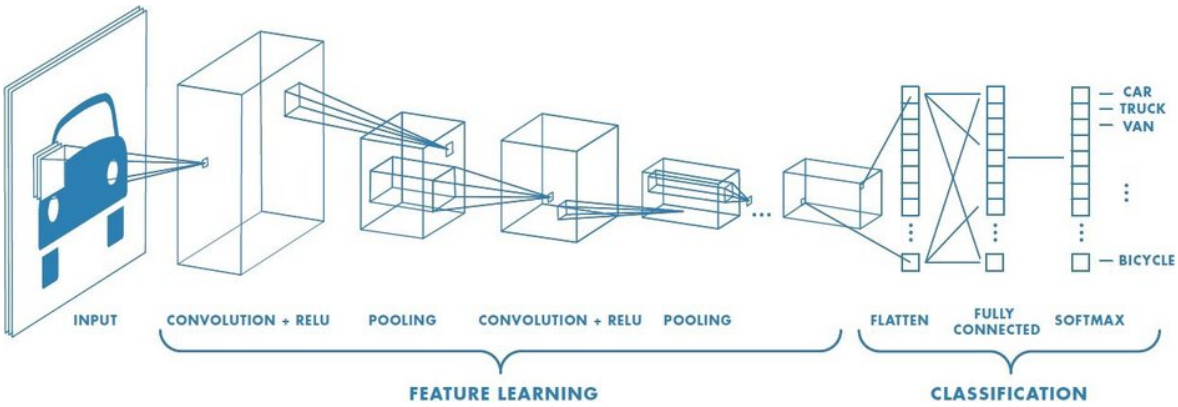


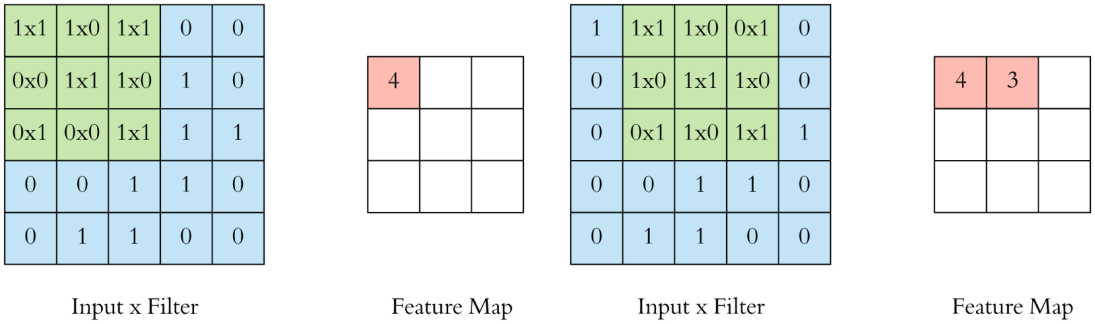
Figura 2.17: Arquitectura de una red neuronal artificial.

Ahora que ya conocemos las ventajas de una CNN frente a una red neuronal ordinaria, podemos profundizar en su arquitectura y comentar cada uno de sus componentes:

Como podemos ver en la figura 2.17, contamos con una fase de extracción de características, formada por neuronas convolucionales y de reducción de muestreo, y otra fase compuesta por neuronas sencillas dedicadas a la clasificación final sobre las características extraídas.

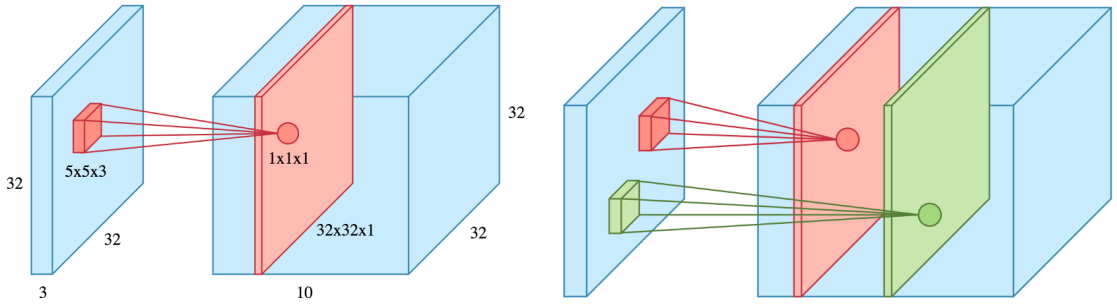
2.4.1 Fase de extracción de características

Según progresan los datos a lo largo de esta fase, se disminuye su dimensionalidad, siendo las neuronas en capas lejanas mucho menos sensibles a perturbaciones en los datos de entrada, pero al mismo tiempo siendo estas activadas por características cada vez más complejas.



(a) Primer paso. **(b)** Segundo paso.

Figura 2.18: Los dos primeros pasos de una convolución.



(a) Dimensiones de los datos **(b)** Acción de dos kernels independientes

Figura 2.19: Convolución sobre una imagen RGB.

Entonces, ante una imagen de entrada, se ejecutan una serie de operaciones de convoluciones y pooling (reducción de muestreo):

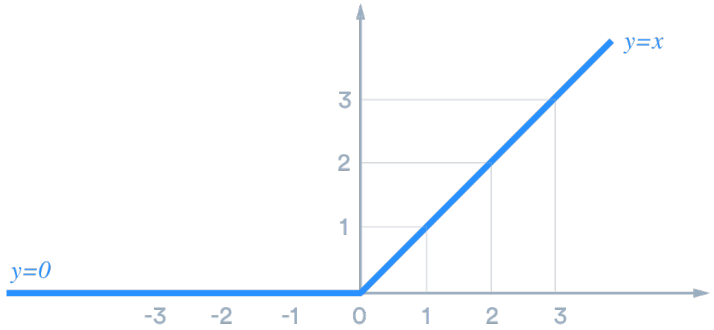


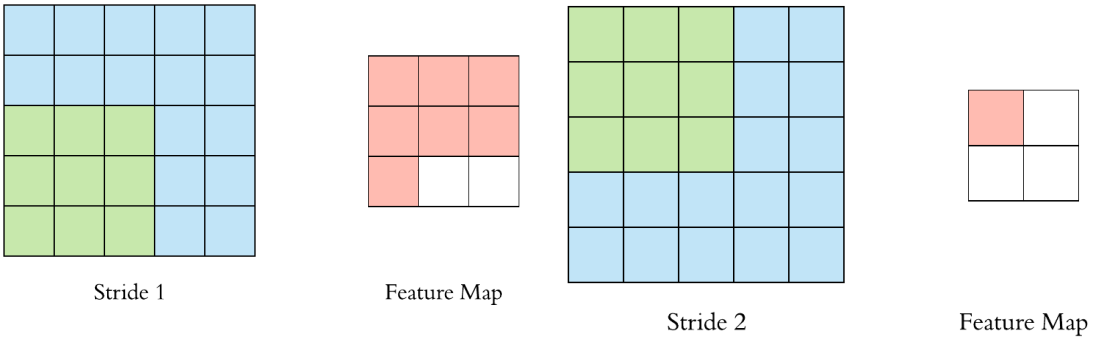
Figura 2.20: Función ReLU: $y = max(0, x)$

Filtro de convolución: Es uno de los componentes principales de una CNN. Consiste en una operación matemática cuya función es combinar dos conjuntos de información. En este caso, la convolución es un filtro que se aplica a la información de entrada para producir mapas de características, sobre los cuales trabajar. Como vemos en la figura 2.18, el filtro, también llamado kernel, se desplaza sobre la imagen de entrada y va construyendo el mapa de características en función de la celda central en cada instante y de sus vecinos. En el ejemplo anterior, se expone una operación de convolución en 2D usando un kernel de 3x3 celdas. No obstante, en realidad dichos kernels son 3D:altura, anchura y profundidad, donde la profundidad son, al igual que en la imagen de entrada, los tres canales de color RGB. La única diferencia será que la suma de los productos de la matriz será en 3D, ya que el resultado de dicha operación seguirá siendo un escalar, una celda dentro del mapa de características resultante. Dicho comportamiento se ilustra en la figura 2.19, donde se muestra la aplicación de diez kernels independientes entre sí a una misma imagen de entrada. En rojo y en verde se muestra la acción de dos de esos diez kernels, los cuales se desplazan a lo largo de toda la imagen. El alto y el ancho de los mapas de características resultantes no varían de tamaño con respecto a la imagen de entrada debido al padding, que será explicado más adelante.

Una vez aplicada la operación de convolución, el resultado es procesado por una **función ReLU**, que es **no lineal**. Véase la figura 2.20.

A continuación, veamos una serie de variables de los kernels que debemos imponer nosotros como desarrolladores. Dichos parámetros, al no ser entrenables, son llamados hiperparámetros:

Stride: Este valor especifica cuántas celdas debe desplazarse el filtro de convolución por cada paso. En el ejemplo de la figura 2.21a dicho valor es 1, como podemos observar. Con, por ejemplo, un stride igual a 2, estaríamos reduciendo aun más el mapa de características resultante. Véase la figura 2.21b. Algunas redes, en lugar de usar la técnica de pooling (véase siguiente punto), hacen uso de este parámetro.



(a) Stride de una celda por paso.

(b) Stride de dos celdas por paso.

Figura 2.21: Ejemplos de strides.

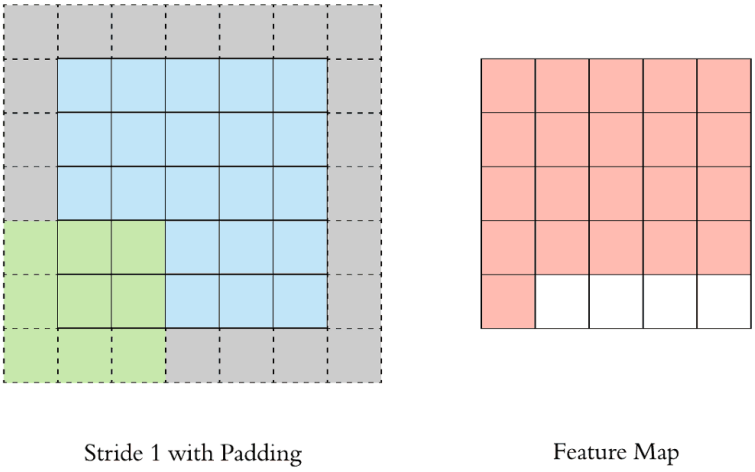
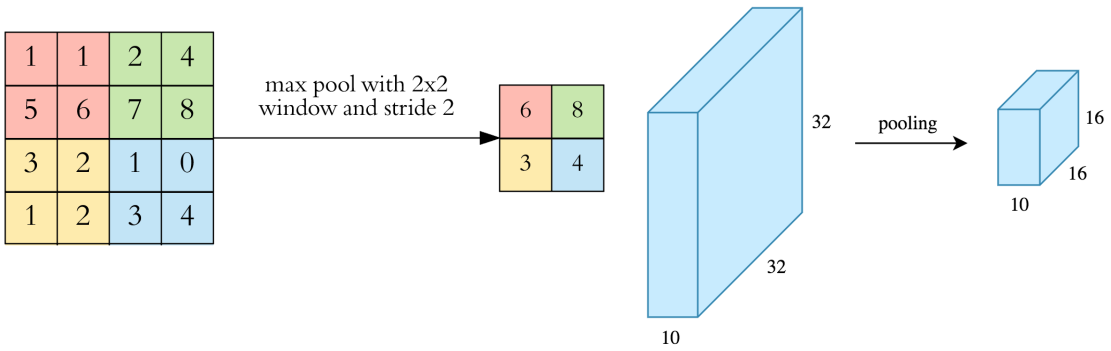


Figura 2.22: Ejemplo de padding.



(a) Simplificación de pooling en 2D.

(b) Resultado de aplicar el pooling al volumen 3D.

Figura 2.23: Ejemplo de pooling.

Padding: Como hemos comentado anteriormente, si no queremos que el alto y el ancho del mapa de características varíe con respecto a la imagen de entrada (o mapa de características de entrada), deberemos recurrir al padding. Dicha técnica consiste en rellenar de ceros los bordes de la imagen para así no reducir en exceso el tamaño de la imagen a lo largo de las múltiples convoluciones que tienen lugar en el proceso de extracción de características. Véase la figura 2.22, donde la zona gris son las celdas de relleno.

Número de filtros por convolución y tamaño: En cuanto al número de filtros, debe ser una potencia de 2 entre 32 y 1024. Cuanto mayor el número de filtros, más potente la red, pero debemos tener cuidado con el overfitting. Por otro lado, el tamaño de dichos filtros suele ser de 3x3.

Filtro de pooling: Tras una operación de convolución, generalmente se lleva a cabo un pooling para reducir la dimensionalidad. Esta reducción de muestreo disminuye el número de pesos, reduciendo los tiempos de entrenamiento y ejecución, combatiendo además el overfitting. Nótese que solo la altura y la anchura se verán reducidas, quedando intacta la profundidad. El procedimiento más habitual es el max pooling, que consiste en mantener el valor máximo dentro de la ventana de pooling. Aunque también se puede hacer la media de dichas celdas, lo cual se conoce como average pooling. En la figura 2.23, podemos ver cómo funciona una ventana de 2x2 celdas con un desplazamiento de 2 celdas por paso. El alto y el ancho del mapa de características, como vemos, se reducen a la mitad. Con ello, conseguimos reducir el número de pesos a la cuarta parte.

2.4.2 Fase de clasificación

Como ya hemos visto, en la fase de extracción de características, la imagen de entrada sufre un proceso de múltiples convoluciones + ReLU + pooling. Tras dicho proceso, obtenemos como salida un volumen 3D, el cual ahora sí es reordenado en un vector de una dimensión e introducido como entrada a una red neuronal ordinaria. Recordemos que la salida de dicha red será, en caso de encontrarse un objeto en la imagen (en nuestro caso, una cara), su clase (contento).

2.4.3 Entrenamiento e inferencia de la red

Al igual que antes, tendremos un proceso de entrenamiento, que será cuando la red aprenda, y otro de inferencia, que será cuando nuestra red se aplique a situaciones desconocidas. En el primer caso, solo haremos forward propagation, mientras que en el segundo, haremos también backpropagation para así ajustar los pesos de la red, tanto de la fase de extracción de características como los pesos de la fase de clasificación.

2.4.4 Niveles de abstracción

Como ya hemos comentado anteriormente, si una red neuronal ha sido correctamente entrenada, sus capas forman una jerarquía de características desde un nivel de abstracción más bajo hasta uno más alto.

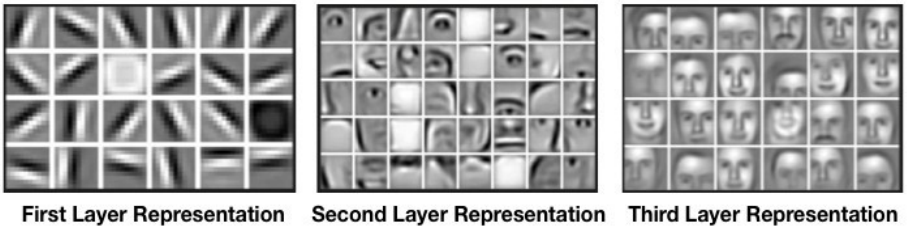


Figura 2.24: Mapas de características de distinta complejidad.

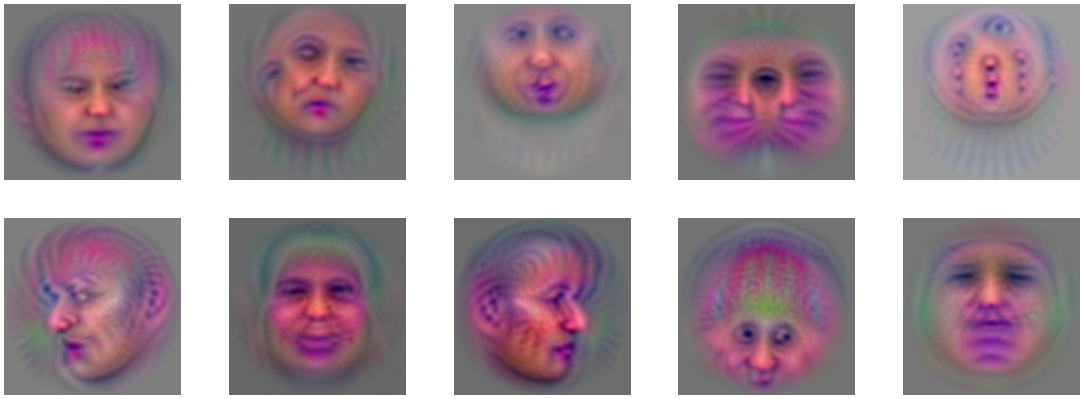


Figura 2.25: Ejemplo de filtros del detector CNN+MMOD para caras.

Este aspecto resulta más intuitivo en la fase de extracción de características, donde los filtros empiezan detectando características más de bajo nivel en sus primeras capas (tales como bordes o esquinas) y acaban detectando patrones más complejos como ojos o bocas en sus últimas capas. Para entender mejor esto último, fijémonos en la figura 2.24.

Resulta también curioso observar algunos filtros convolucionales reales del detector CNN+MMOD de Dlib para caras, del cual hablaremos en la siguiente sección. Véase la figura 2.25.

2.5 Detección de objetos

2.5.1 Esquema general de un algoritmo de detección

La arquitectura antes expuesta funciona recibiendo como entrada una imagen en la cual debe aparecer centrado el objeto a reconocer y con una determinada escala y ratio de aspecto. Resuelve, por tanto, problemas de clasificación.

No obstante, para nuestra aplicación, tendremos que interpretar frames en los que la cara de la persona en cuestión vaya variando de localización y de escala con el tiempo. Es por ello que nos enfrentamos a un problema no de clasificación, sino de detección.

Veamos, entonces, el algoritmo habitual de un detector de objetos:

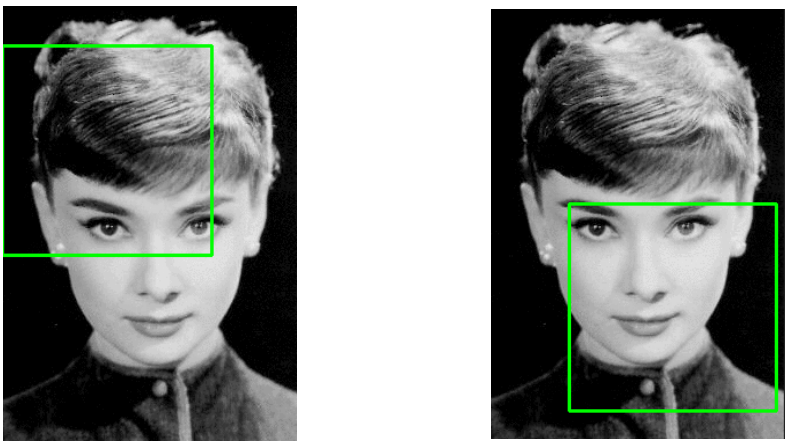


Figura 2.26: Ejemplos de ventanas.

1. **Extracción de regiones de interés:** En primer lugar, se aplican ventanas de escala y ratio de aspecto diferentes, las cuales van recorriendo la imagen. Véase la figura 2.26. Lógicamente, se da un compromiso entre el número de ventanas aplicadas y la complejidad computacional. Cuantas más ventanas recorran la imagen, mayor será la probabilidad de encontrar objetos de distinto tamaño, pero mayor será también el coste computacional.

2. **Clasificación de cada región sustraída:** De cada una de las regiones locales obtenidas, se extraen las características más representativas, las cuales determinan la clase a la cual pertenece la subimagen. Este paso es realizado por la red neuronal convolucional.

3. **Supresión de detecciones solapadas:** Por como funciona el algoritmo, se detectará el mismo objeto múltiples veces, por lo que en este paso debemos descartar aquellas detecciones que no sean la mejor para un mismo objeto.

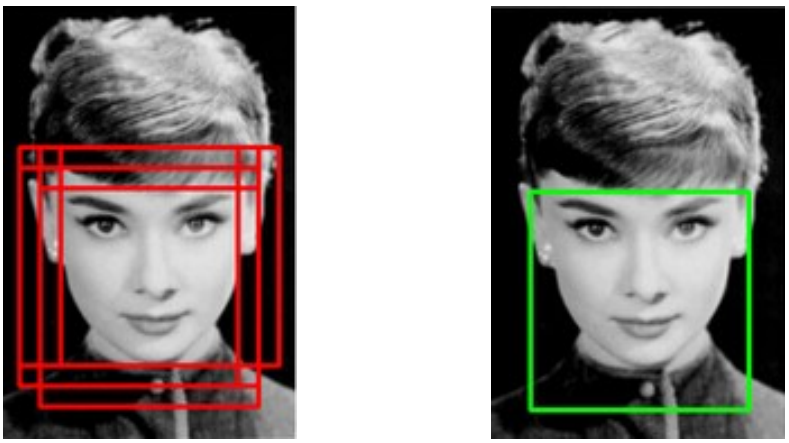


Figura 2.27: Antes y después de aplicar Non-Maximum Suppression.

Para ello, todas las cajas superpuestas son combinadas en una sola mediante Non-Maximum Suppression. Véase la figura 2.27. No obstante, este paso también puede que descarte aquellos objetos que aparezcan solapados por otros.

Finalmente, el resultado debe ser que todos aquellos objetos de la imagen que queramos detectar y clasificar, estén encuadrados y tengan asociadas unas coordenadas espaciales y una probabilidad de acierto.

2.5.2 Algoritmo de detección CNN+MMOD

Mediante el procedimiento descrito anteriormente, un conjunto de ventanas de imágenes positivas y negativas son seleccionadas del conjunto de entrenamiento. Entonces, un clasificador es entrenado en estas ventanas. Por último, el clasificador se prueba en imágenes que no contengan objetos de interés, identificándose ventanas de falsa alarma, las cuales son añadidas al conjunto de entrenamiento. Entonces, el clasificador es reentrenado y, opcionalmente, este proceso se itera.

Dicho enfoque no hace un uso eficiente de los datos de entrenamiento al entrenar solamente con un subconjunto de ventanas. Además, las ventanas que solapan un objeto son una fuente común de falsos positivos. Este procedimiento de entrenamiento hace difícil incorporar directamente estos ejemplos en el conjunto de entrenamiento, ya que dichas ventanas no son ni totalmente un falso positivo ni una detección correcta. Y lo que es más importante, la precisión del sistema de detección de objetos como un todo, no está siendo optimizada. En su lugar, la precisión del clasificador sobre el subconjunto de entrenamiento es usada como aproximación.

No obstante, el optimizador CNN+MMOD usado por dlib se ejecuta sobre todas las ventanas y optimiza el rendimiento de un sistema de reconocimiento de objetos en función del número de detecciones omitidas y falsos positivos a la salida final del sistema. Además, la formulación de dicho descriptor consigue una optimización convexa y emplea un algoritmo que encuentra el conjunto de parámetros globales más óptimos.

Para ello, se utiliza una capa de coste MMOD que puede usarse para aprender un detector de objetos de un conjunto de entrenamiento con cajas de toda forma y tamaño. Para ilustrar todo esto, el creador del detector, Davis E. King, escribe un código capaz de detectar vehículos.

Dicho detector está entrenado con un conjunto de apenas 2217 imágenes, bastante pocas para lo habitual.

El esquema que sigue el algoritmo de detección CNN+MMOD de Dlib es el siguiente:

1. Creamos una pirámide de imágenes, la cual se introduce en una única imagen de grandes dimensiones. Llamemos a esto la "pirámide apilada".

2. Ejecutamos sobre dicha pirámide apilada una CNN. La CNN devuelve como salida una nueva imagen donde los píxeles de mayor intensidad indican la presencia de vehículos.

3. Localizamos los píxeles de la imagen cuyo valor de intensidad sea mayor que 0. Dichas localizaciones serán detecciones preliminares de vehículos.

4. Ejecutamos NMS sobre las detecciones preliminares, produciendo la salida final.

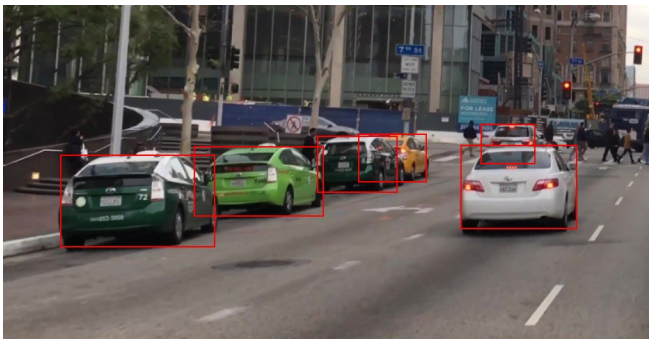


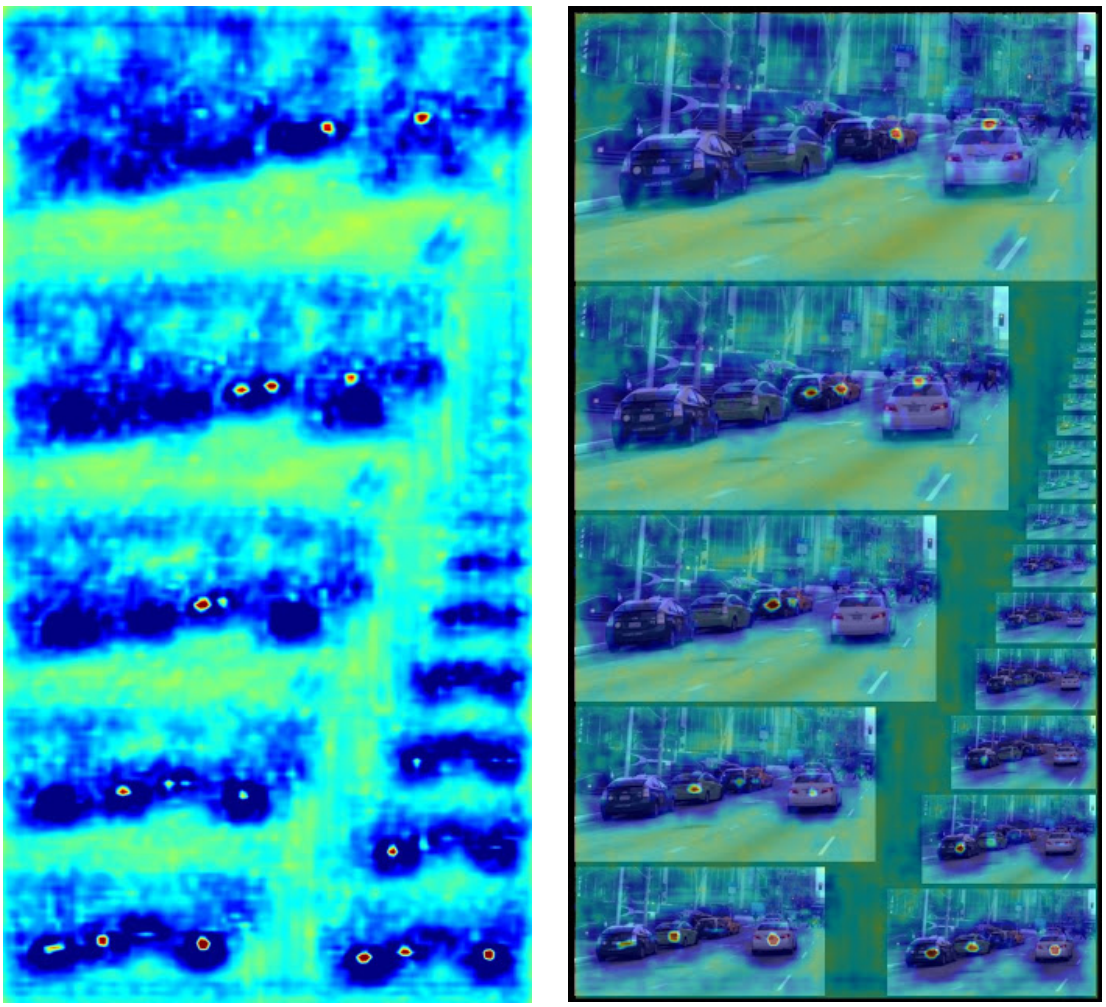
Figura 2.28: Entrada de ejemplo al detector CNN+MMOD.

Profundicemos en los dos primeros pasos y veamos el ejemplo de la figura 2.28:

En primer lugar, debemos crear lo que hemos llamado pirámide apilada, véase la figura 2.29. La CNN la recibe como entrada, le aplica una serie de convoluciones y devuelve como salida un nuevo conjunto de imágenes. En el caso del detector de vehículos, se devuelven tres nuevas imágenes, siendo cada una de ellas un mapa de intensidades de detección que destaca aquellas localizaciones donde más probablemente se encuentre un vehículo. La razón por la que hay

Figura 2.29: Entrada de ejemplo al detector CNN+MMOD.

tres imágenes es porque podemos decir que hay tres ratios de aspecto diferentes; uno para los camiones, otro para los turismos, y el último ratio para los todoterreno; para los cuales se usan distintas ventanas deslizantes. Para visualizar mejor los resultados, las tres imágenes se muestran combinadas en la figura 2.30a. Por otro lado, el propio detector puede decidir utilizar múltiples ventanas deslizantes automáticamente. No obstante, en nuestro caso, al querer detectar caras, solo necesitaremos una única ventana deslizante de un determinado ratio de aspecto.



(a) Salidas combinadas. **(b)** Pirámide apilada y salidas.

Figura 2.30: Pirámide apilada con las detecciones obtenidas.

Figura 2.31: Salida final del detector.

Volviendo al ejemplo del detector de vehículos, si superponemos las salidas de la red sobre la pirámide apilada, podemos comprobar que funciona correctamente. Observemos además, que los coches más pequeños son detectados en la parte superior de la pirámide, mientras que conforme la bajamos se van detectando aquellos vehículos de mayor escala. Véase la figura 2.30b. Hemos, por tanto, conseguido que nuestro detector sea invariante a escala sin utilizar ventanas deslizantes de distinta escala.

Una vez la red obtiene una salida, lo único que el algoritmo debe hacer es umbralizar la salida, encontrar todas las zonas probables de localización, aplicar NMS y devolver a su salida las cajas correspondientes con la localización de los vehículos.

Por otro lado, uno de los aspectos más importantes de este detector es la capa de coste MMOD, la cual cuenta, durante el proceso de entrenamiento, el número de errores de detección (falsos positivos, detecciones omitidas y detecciones duplicadas), incluyéndolo en la fase de propagación hacia atrás. Además, como dicha capa de coste cuenta los errores tras haberse aplicado NMS, la red aprende que debe evitar producir zonas de alta probabilidad de localización en aquellas zonas de la imagen donde NMS no va a poder actuar. Esta es la razón por la cual en la figura 2.31 veíamos las zonas en azul oscuro, que indican donde claramente no encontramos ningún vehículo, rodeando cada una de las detecciones. La red consigue aprender, en conclusión, la delgada línea entre "hay un coche" y "no hay un coche", evitando así detectar el mismo vehículo más de una vez.

En definitiva, el detector CNN+MMOD de Dlib necesita una menor cantidad de datos de entrenamiento (al hacer un uso más eficiente de estos), su coste computacional es relativamente bajo (al no tener que procesar múltiples ventanas deslizantes de distinto tamaño ni, en nuestro caso, de distinto ratio de aspecto), y es más eficaz (al conseguir una optimización global de los parámetros, gracias al empleo de la capa de coste MMOD).

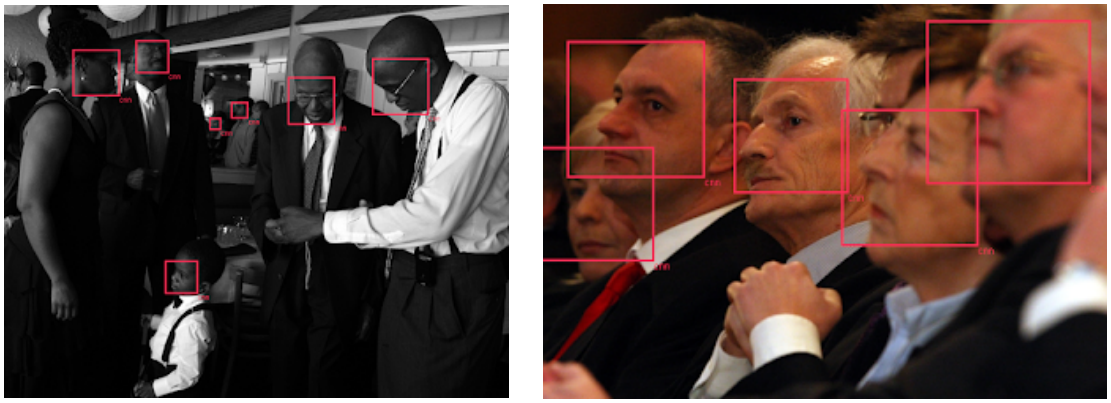


Figura 2.32: Resultados de aplicar CNN+MMOD a la detección de caras.

Como una imagen vale más que mil palabras, véase que en los ejemplos de la figura 2.32 se reconocen correctamente las caras a pesar de aparecer ocluidas o tener una escala muy reducida.

3 Propuesta

El ocio y el juego son vitales en el desarrollo personal de un niño. Le enseñan a cómo comportarse en sociedad, le enfrentan a desafíos parecidos a los que pueden encontrarse en la edad adulta, permiten que establezca lazos de amistad con otros niños, etcétera. No obstante, aquellos con necesidades especiales, como pueden ser los niños con autismo, experimentan dificultades a la hora de relacionarse y jugar con sus compañeros.

Varios trabajos pretenden mejorar las capacidades sociales de los niños con autismo mediante la integración de la robótica social en las terapias existentes. Un ejemplo es (Shah Johan Hamzah y cols., 2014), en el que se programa un robot NAO para que sirva como herramienta de aprendizaje, pudiendo mantener diálogos con el niño. Por otro lado, (Robins y cols., 2004) proponen que el robot sirva como fuente de atención compartida entre el niño con autismo y el terapeuta, o entre el niño con autismo y otros niños, propiciando la interacción entre estos. Otro ejemplo interesante es (Vanderborght y cols., 2012), que utiliza el robot social Probo para contar cuentos sociales a niños con trastorno del espectro autista.

En este contexto, nuestra propuesta pretende mejorar las habilidades sociales de los niños, especialmente de aquellos con autismo, teniendo en cuenta su posible diversidad funcional. Para ello, se diseña una aplicación que permite al robot Pepper contar un cuento social que, además, sea flexible e interactivo. Dicha aplicación cuenta con tres módulos bien diferenciados:

- **El reconocimiento de emociones**, para monitorizar el estado de ánimo de la persona en todo momento y actuar en consecuencia. Este lo conforman el detector de caras y el clasificador de emociones.

- **La aplicación web**, que será la responsable de mostrar las distintas escenas de un cuento a través de la tablet del robot. Mostrará no solo el texto, sino también imágenes que refuercen el relato.

- **El programa principal**, el cual nos permitirá indicarle al robot Pepper cómo deberá comportarse en cada instante, además de coordinar el funcionamiento de la aplicación en su conjunto.

3.1 Detector de caras

3.1.1 Justificación del detector escogido

Antes de decantarnos por el detector de caras CNN+MMOD de Dlib, otros métodos fueron analizados y tomados en consideración. De entre los diversos detectores existentes, se han comparado el Haar Cascade de OpenCV, el detector DNN de OpenCV, el de HoG+MMOD de Dlib y el propio CNN+MMOD. Para ello, nos basamos en el trabajo presentado en (Gupta, s.f.).

Características principales: A continuación, detallamos los aspectos principales de cada uno de los detectores antes mencionados:

- **Haar Cascade de OpenCV:** Dicho detector funciona casi en tiempo real cuando se ejecuta en una CPU, es capaz de detectar caras a distintas escalas y tiene una arquitectura bastante simple. No obstante, no es un detector nada preciso, ya que devuelve un número considerable de falsos positivos y no funciona con caras ocluidas o que no tengan una pose totalmente frontal.

- **DNN de OpenCV:** Además de funcionar en tiempo real cuando se ejecuta en una CPU, el detector DNN de OpenCV es el más preciso de los detectores comparados. Esto último se debe a que funciona ante distintas poses u orientaciones de las caras, es robusto a oclusiones y detecta un amplio rango de escalas.

- **HoG+MMOD de Dlib:** Este detector es bastante eficiente,y funciona bien con poses que no son del todo frontales y pequeñas oclusiones. Sin embargo, al ser entrenada con caras de mínimo 80x80 píxeles, aquellas con menor escala no serán detectadas. No obstante, podríamos reentrenar el detector con caras más pequeñas para solventar este problema. Por otro lado, tampoco funciona excesivamente bien ante grandes oclusiones o poses complicadas.

- **CNN+MMOD de Dlib:** Como ya se comentó en el apartado 2, dicho detector funciona muy bien ante distintas poses y orientaciones de las caras, y es robusto ante oclusiones. Además de contar con una alta precisión, es muy eficiente si se ejecuta en una GPU. Por otro lado, el proceso de entrenamiento es bastante sencillo. No obstante, es demasiado lento en una CPU y, al igual que con el método HoG+MMOD, este detector ha sido entrenado con caras de mínimo 80x80, por lo que para detectar caras de menor escala tendríamos que reentrenar la red con ejemplos de menor tamaño.

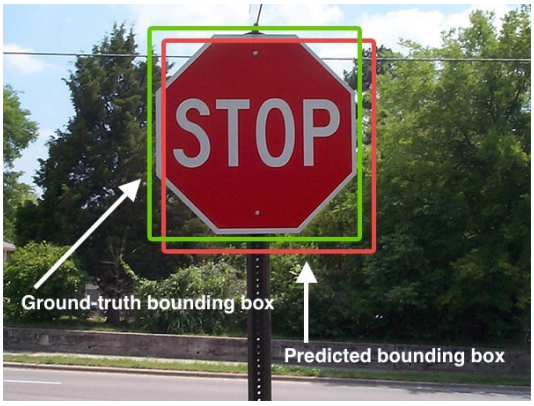


Figura 3.1: Encuadre esperado (verde) y encuadre obtenido (rojo).

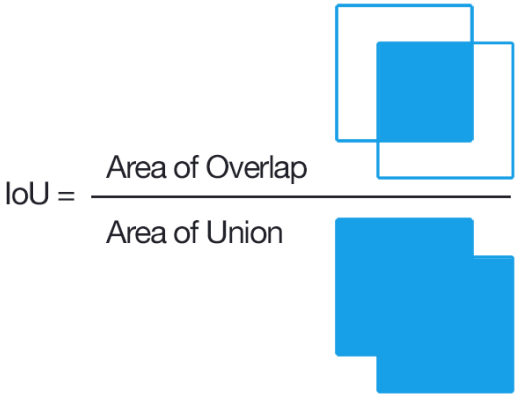


Figura 3.2: Intersección sobre la unión.

Comparativa de eficacia: Seguidamente, se expondrán las métricas usadas para comparar la precisión de los métodos citados y se comentarán los resultados para cada uno de ellos:

- **Métricas empleadas:**

  – En primer lugar, contamos con el método de **Intersección sobre Unión (IoU)**. Dicha métrica se utiliza para determinar cómo de acertado es el encuadre del objeto detectado. Es decir, cuánto se acerca el encuadre obtenido por el detector en cuestión al encuadre real fijado en el conjunto de datos. Véase la figura 3.1. Como contamos con información acerca del encuadre predicho y el real, simplemente debemos calcular el área de solape entre ambos encuadres dividido por el área de unión. Véase la figura 3.2.

  Lógicamente, el valor obtenido por la métrica IoU debe ser un número decimal comprendido entre el 0 y el 1, que representa el ratio del área solapada. Cuanto más se acerce dicho valor al 1, más preciso será el encuadre proporcionado por el detector empleado.

– El otro método usado para medir la precisión de los detectores es la **Precisión Media (Mean Averaged Precision, mAP)**. Dicha métrica determina la capacidad de un detector de detectar correctamente y clasificar un objeto con respecto a su clase. En el caso de detección de caras, que es el que nos incumbe, medimos cómo de correcto es un detector a la hora de detectar y clasificar un objeto como cara, y no como fondo. Dicho valor se calcula de la siguiente manera:

$$mAP = \frac{PR}{PR+FP}$$

Lo que hemos llamado *PR (Positivos Reales)* representa el número de casos en los cuales se ha detectado correctamente una cara, mientras que los *FR (Falsos Positivos)* son aquellas áreas predichas como caras que en realidad son fondo.

En el campo de detección de objetos, caras en nuestro caso, normalmente se determinan primero aquellos casos en los que el valor de IoU supere un cierto umbral. Entonces, para dichos casos, se calcula el valor de AP. Por ejemplo, una precisión de 0.8 para un AP@50 es el resultado de calcular la precisión AP para aquellos encuadres en los que el IoU sea superior al 50%.

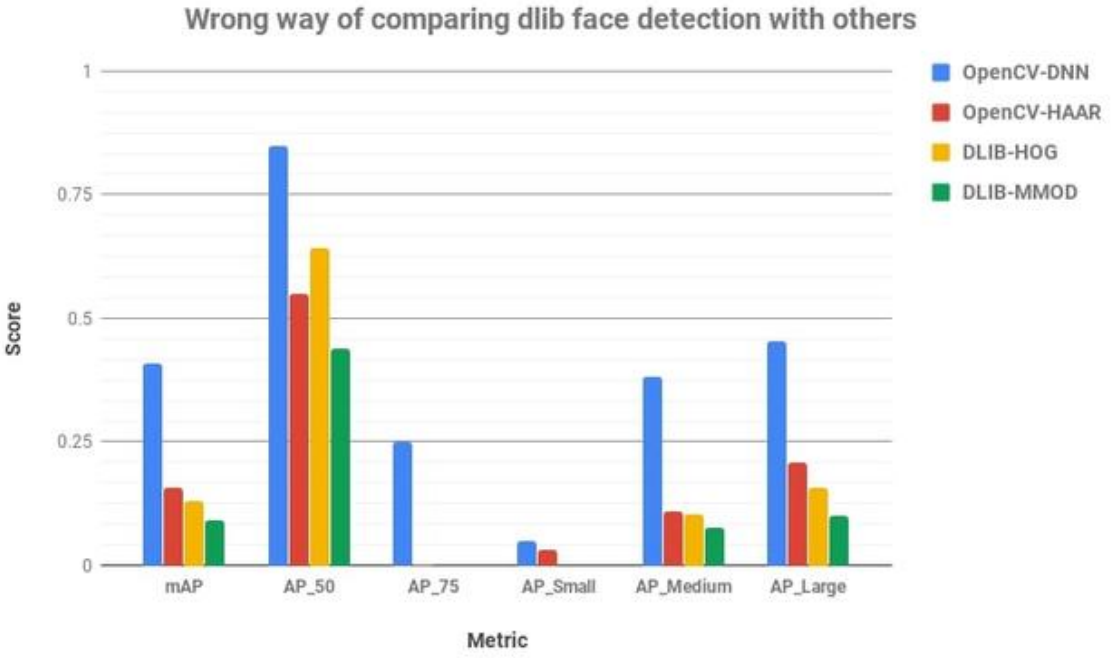


Figura 3.3: Comparativa en eficacia de los detectores.

- **Resultados obtenidos:**

  Una vez comentadas las métricas empleadas, podemos proceder a analizar los resultados obtenidos para cada detector de caras, siendo evaluados frente al conjunto de datos FDDB (Face Detection Data set and Benchmark). Véase la figura 3.3, donde:

  - **AP_50**: Precisión media para aquellos casos en los que el solape entre encuadres sea superior al 50%.
  - **AP_75**: Precisión media para aquellos casos en los que el IoU sea superior al 75%.
  - **AP_Small**: Precisión Media para caras de escala pequeña (IoU entre el 50 y el 95%).
  - **AP_Medium**: Precisión Media para caras de escala media (IoU entre el 50 y el 95%).
  - **AP_Large**: Precisión Media para caras de escala grande (IoU entre el 50 y el 95%).
  - **mAP**: Precisión Media para todos los IoU entre el 50 y el 95%.

  Curiosamente, el nivel de precisión obtenido por el detector CNN+MMOD de Dlib es menor que con Haar Cascade. No obstante, intuitivamente podemos comprobar que con CNN+MMOD se detectan muchas caras que con Haar Cascade, no. Véase el ejemplo de la figura 3.4.

  Por tanto, podemos afirmar que los malos resultados de eficacia obtenidos por CNN+MMOD no son representativos. Esto se debe principalmente a que dlib fue entrenado usando conjuntos de entrenamiento estándar, pero con las anotaciones hechas por el propio Davis King. Por ello, las cajas obtenidas son distintas de las establecidas en el conjunto de entrenamiento, lo cual reduce el IoU. Véase como en la figura 3.5, la frente y la barbilla de la persona caen fuera de la caja. Otra de las razones es que, recordemos, Dlib no es capaz de detectar caras cuya escala sea inferior de 80x80 píxeles. Fijémonos que para AP_small los métodos de Dlib directamente desaparecen de la tabla.

  En definitiva, AP_50 sería la medida más adecuada de las expuestas, pero sigue sin ser del todo representativa de la realidad.

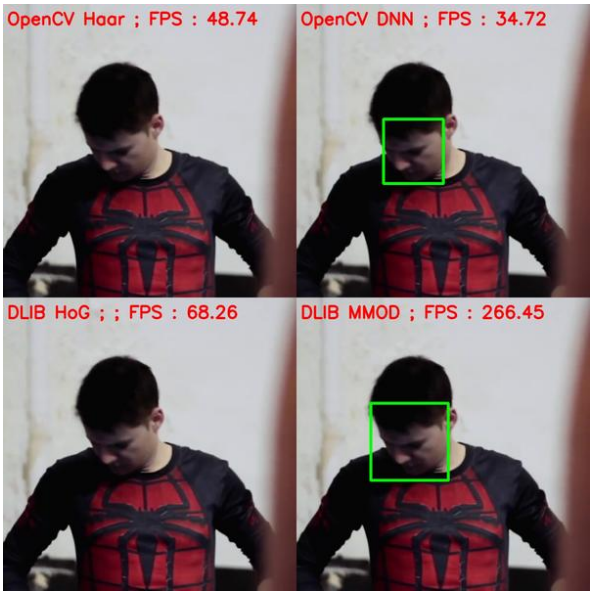


Figura 3.4: Ejemplo de pose no frontal.

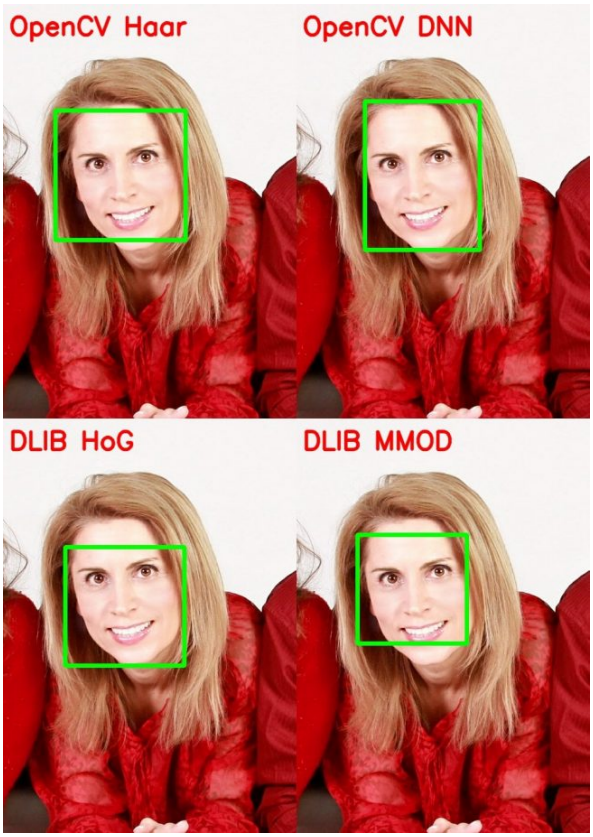


Figura 3.5: Ejemplo de encuadres con los detectores.

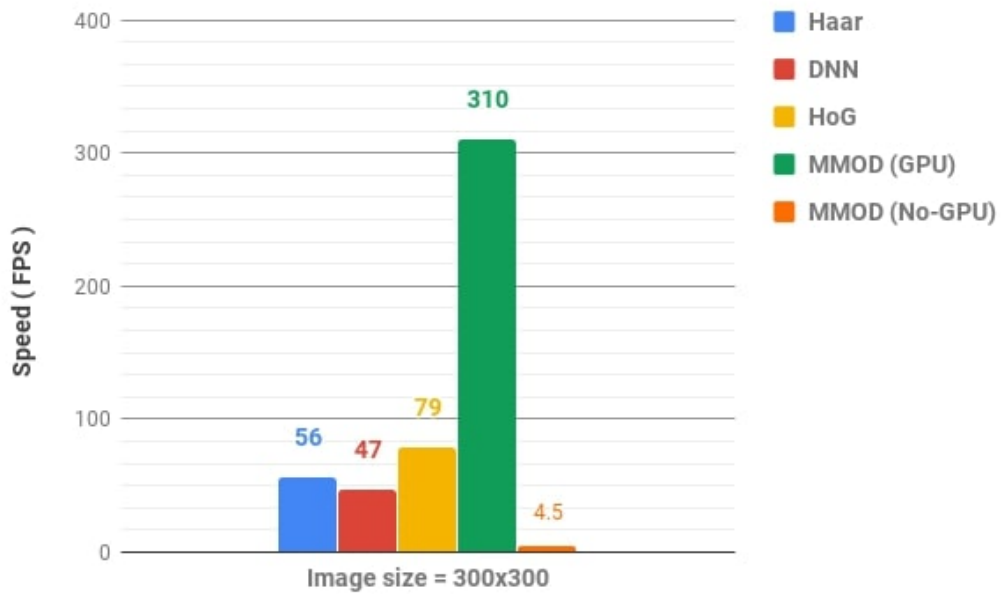


Figura 3.6: Comparativa en eficiencia de los detectores.

Comparativa de eficiencia: El rendimiento de un detector no viene determinado únicamente por su eficacia, sino que hay que tener en cuenta también la velocidad con la que funciona, sobre todo para problemas de tiempo real, como es el nuestro. A continuación, se detallan las condiciones bajo las cuales se llevan a cabo las pruebas y se analizan los resultados obtenidos para cada detector.

- **Condiciones de las pruebas:**

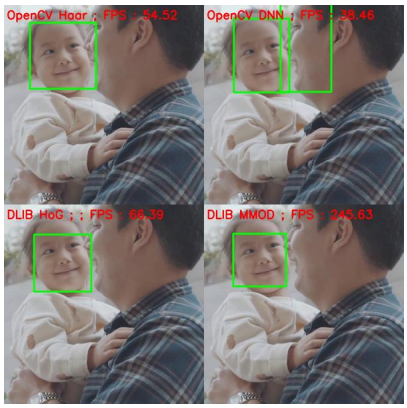
  El hardware empleado es el siguiente:

  - Procesador: Intel Core i7 6850K - 6 Core.
  - Memoria RAM: 32 GB.
  - GPU: NVIDIA GTX 1080 Ti with 11 GB RAM.
  - Sistema Operativo: Linux 16.04 LTS.
  - Lenguaje de programación: Python.

  Cada detector se ejecuta 10000 veces para una imagen dada de 300x300 píxeles y se calcula el tiempo tardado en cada iteración por cada detector.

- **Resultados obtenidos:**

  Como puede verse en la figura 3.6, tanto Haar, como DNN y HoG funcionan en tiempo real en una CPU, mientras que el detector CNN+MMOD es demasiado lento para ello. No obstante, al ser ejecutado en una GPU, el método más rápido con diferencia es el detector CNN+MMOD. Los detectores de OpenCV solo pueden ser ejecutados en una CPU debido a que OpenCV no ha adoptado todavía el framework de NVIDIA para ello.



(a) Ejemplo de pose no frontal.

(b) Ejemplo de cara ocluida.

Figura 3.7: Ejemplo de caras complicadas de detectar.

En definitiva, tanto el detector DNN de OpenCV como el CNN+MMOD de Dlib cuentan con una precisión muy similar. Es posible que el primero sea algo más robusto a oclusión o poses de caras distintas a la frontal, además de ser capaz de detectar caras de menor tamaño. Véanse las figuras 3.7a y 3.7b. No obstante, esto no es determinante para nuestra aplicación, puesto que el niño en cuestión estará a una distancia reducida del robot y mirándole normalmente a la cara, donde se encuentra la cámara que toma los frames a analizar.

Además, contamos con una GPU, por lo que la solución más sensata es la de utilizar el detector **CNN+MMOD de Dlib**.

3.1.2 Arquitectura de la red

Una vez justificada nuestra elección, podemos proceder a ahondar en la arquitectura del detector CNN+MMOD de Dlib.

Figura 3.8: Filtros básicos.

Bloques principales: A continuación, comentaremos las distintas transformaciones que conforman la red.

En primer lugar, se definen dos filtros convolucionales mostrados en la figura 3.8:

- **Con5:** Cuyo tamaño es también de 5x5 celdas y tiene un stride de 1x1.
- **Con5d:** Cuyo tamaño es de 5x5 celdas y que cuenta con un stride de 2x2, lo cual reducirá la dimensionalidad del mapa de características sobre el cual se aplique.

Posteriormente, se implementan dos conjuntos de capas bien diferenciados, que hemos denominado **rcon5** y **downsampler**:

- **Rcon5:** Consiste simplemente en aplicar una capa convolucional de 45 filtros independientes entre sí, cuyo tamaño es de 5x5 y cuyo stride es de 1x1, por lo que no se reduce la dimensionalidad del mapa de características. La salida de este conjunto de filtros es procesada posteriormente por una capa afín+ReLu, la cual permite a la red aprender combinaciones no lineales de las características extraídas dentro de cada filtro de la capa convolucional. Digamos que, si los filtros convolucionales aprenden piezas de puzzle (características locales), la capa afín + ReLu ayuda a juntarlas todas y darles un contexto, una visión general dentro del mapa de caraterísticas.

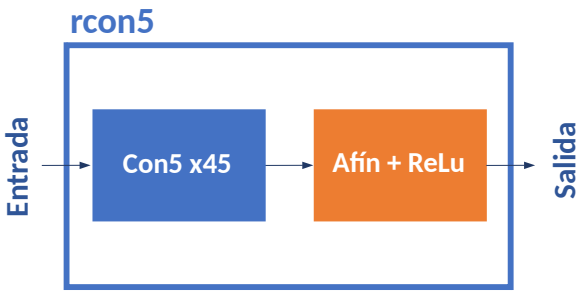


Figura 3.9: Bloque rcon5.

- **Downsampler:** En vez de usar capas de pooling, se diseña un bloque de reducción de dimensionalidad, el cual llamamos downsampler. Dicho bloque consiste en un conjunto de capas **con5d**, que, recordemos, tiene un stride de 2x2 celdas. Al haber tres de estas capas, la imagen de entrada se reduce x8.

  Esta reducción de dimensionalidad no solo mejora la eficiencia de la red, al reducir el número de operaciones necesarias, sino que también es aquello que le confiere la capacidad de abstracción. Esto último se debe a que, reduciendo la dimensionalidad del mapa de características, se compactan dichas características en otras más generales, que serán procesadas en capas superiores y nuevamente reducidas a características cada vez más abstractas.

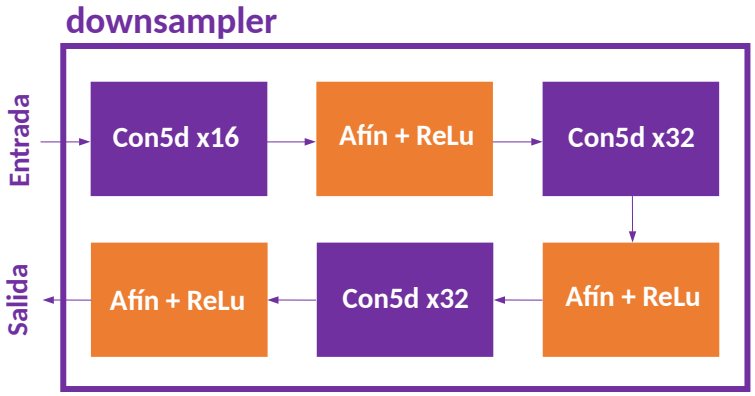


Figura 3.10: Bloque de reducción de dimensionalidad.

Concepto de profundidad: Como podemos observar, las capas convolucionales se componen de múltiples filtros independientes entre sí. Por ejemplo, dentro del bloque **rcon5**, se emplean 45 filtros convolucionales seguidos. Dichos filtros extraen distintas características de las mismas zonas locales de la imagen de entrada. Es decir, mientras que uno de los filtros puede que se active ante la presencia de bordes de una determinada orientación, otro filtro lo hará ante bordes de una orientación distinta.

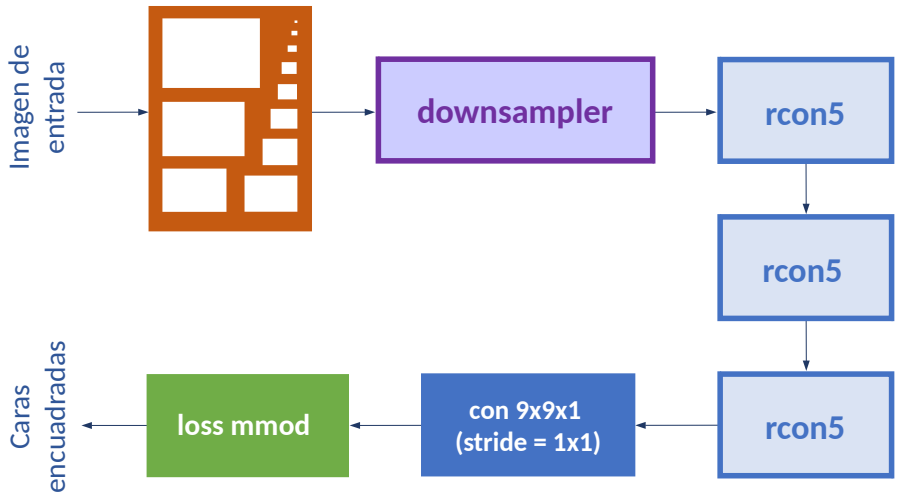


Figura 3.11: Estructura general del detector.

En un nivel de abstracción más alto, podríamos decir que cada filtro se activa ante caras de distintas poses, por lo que todas ellas son detectadas sin la necesidad de ejecutar un detector para caras que miran al frente, otro para caras que miran a la derecha, etc. De hecho, los filtros mostrados por la figura 2.25 representan los distintos filtros aplicados por la última capa **rcon5** de la red.

Una convolución grande vs. varias convoluciones pequeñas: Si nos fijamos en la figura 3.11, se aplica tres veces seguidas el bloque **rcon5**. O sea, tres capas convolucionales 5x5 con dos capas afín+ReLu de por medio. Posiblemente nos estemos preguntando si no sería lo mismo aplicar tan solo una gran capa convolucional de 13x13. Para este caso particular, cada neurona de la primera capa comprende 5x5 celdas de la entrada. Después, una neurona de la segunda capa comprendrá 5x5 celdas de la salida de la primera capa, por lo que dispondrá de una vista de 9x9 celdas de la imagen de entrada. Finalmente, la tercera capa comprendrá 5x5 celdas de la segunda capa y 13x13 de la imagen de entrada. Podríamos, entonces, pensar que bastaría con aplicar un filtro de 13x13 y el resultado no cambiaría. No obstante, aunque sí tendríamos el mismo campo de visión de la imagen de entrada, el resultado no sería exactamente el mismo.

En primer lugar, las neuronas de una capa 13x13 estarían aplicando una sola función lineal a la entrada, mientras que un conjunto de tres capas 5x5, con ReLu de por medio, contendrían no-linearidades que dotarían de mayor expresividad a las características devueltas. En segundo lugar, sabiendo que los mapas de características devueltos por dichas capas tienen un número de canales C=45, una sola convolución 13x13 necesitaría Cx(13x13xC) = $169C^2$ parámetros, mientras que tres capas 5x5 apiladas tan solo necesitarían 3x(Cx(5x5xC)) = $75C^2$.

En definitiva, utilizando múltiples capas convolucionales de tamaño reducido consegui-

mos características más representativas con, además, un menor número de parámetros.

No obstante, si volvemos a fijarnos en la figura 3.11, vemos que la última capa de convolución tiene un tamaño de 9x9 celdas. Lo más lógico según el razonamiento anterior, sería pensar que debemos cambiar dicho filtro por, por ejemplo, dos filtros convolucionales 5x5 con una capa afín+ReLu de por medio. Conseguiríamos, en principio, características más expresivas usando tan solo 2x(Cx(5x5xC)) = 50 parámetros frente a Cx(9x9xC) = 81, comprendiendo en ambos casos 81 celdas del mapa de características de entrada.

No obstante, en determinadas ocasiones, como esta misma, es más conveniente usar un solo filtro de mayor tamaño. Esto se debe a que los 50 parámetros, usados por dos filtros 5x5 para describir una zona de 81 celdas, están relacionados y, por tanto, restringidos entre sí, En cambio, los 81 parámetros obtenidos por una sola capa 9x9 son independientes entre sí y proporcionan una mayor libertad de representación para las 81 celdas de entrada.

Por ello, como las características del mapa de entrada son mayores que 5x5, resulta más apropiado tener una capa de 9x9 para representar con mayor libertad dichas características. Si en su lugar empleáramos dos capas 5x5, las características finales 9x9 dependerían de combinaciones 5x5 de las características intermedias obtenidas por la primera capa. En definitiva, existiría una restricción a la hora de distinguir zonas 9x9 entre sí, debido a que estas dependerían de zonas iniciales con un patrón repetitivo 5x5.

Estructura general de la red: Una vez hemos profundizado en los aspectos más concretos de la red, podemos describir de una manera más general la estructura y el funcionamiento del detector.

En primer lugar, recibimos como entrada la pirámide apilada construida a partir de la imagen a analizar. A continuación, el downsampler procesa dicha pirámide extrayendo aquellas características de más bajo nivel y reduciendo x8 la dimensionalidad del mapa de características. Más tarde, la salida es procesada por tres bloques rcon5 que devuelven características cada vez más abstractas, tales como orejas, bocas o directamente caras.

Después, contamos con una última capa convolucional que recibe como entrada un mapa de características 3D y devuelve una imagen de un solo canal de profundidad donde los píxeles de mayor intensidad se corresponderán con las caras detectadas.

Por último, se observa una capa de coste MMOD, que, como comentamos en la sección 2, permite en la fase de entrenamiento una optimización global de los parámetros, descartando, por ejemplo, aquellas caras detectadas que no aparezcan centradas en la ventana, evitando así múltiples detecciones de una misma cara.

Figura 3.12: Muestra de la dataset de entrenamiento del detector.

3.1.3 Entrenamiento y eficacia del detector

El detector CNN+MMOD de Dlib ha sido entrenado mediante un conjunto de entrenamiento de 6975 caras. Dicho conjunto de entrenamiento consiste en una colección de imágenes de caras seleccionadas de otros múltiples conjuntos de entrenamiento disponibles en la red (excluyendo FDDB). En particular, hay imágenes procedentes de ImageNet, AFLW, Pascal VOC, VGG, WIDER y Facescrub. A diferencia de FDDB, este conjunto de entrenamiento contiene caras con un amplio rango de poses, y no solo tomas frontales. En la figura 3.12 pueden verse algunas de las caras del conjunto de entrenamiento recortadas y apiladas en una gran imagen.

Por otro lado, como ya hemos comentado con anterioridad, fijamos el tamaño mínimo para que una cara sea detectada en 80x80 píxeles. En función de esto, el algoritmo MMOD establecerá de manera automática los parámetros de altura y anchura de la ventana deslizante, además de unos parámetros razonables para el Non-Maximum Suppression. A su vez, se perturban los colores de las imágenes de entrenamiento para conseguir una mejor generalización

Una vez entrenado, el detector alcanza una reseñable tasa de acierto de **0.879134**. No obstante, se producen **90 falsos positivos**, lo cual resulta alarmante. Sin embargo, si nos fijamos en la figura 3.13, que muestra los falsos positivos que comentábamos, descubrimos que todos ellos, excepto uno, son realmente caras.

Por último, resulta interesante comentar que el creador del detector, Davis King, fue capaz de entrenar un modelo de esta red con tan solo 4 imágenes de entrenamiento con caras anotadas. A pesar de que cualquier detector necesita cientos de imágenes para demostrar un rendimiento aceptable, dicho modelo consiguió aprender a detectar caras. Véase un ejemplo en la figura 3.14.

Figura 3.13: Supuestas falsas alarmas devueltas por el detector.

Figura 3.14: Demostración del detector entrenado con solo cuatro imágenes.

3.1.4 Integración del detector en nuestra aplicación

Dentro de nuestra aplicación, creamos un hilo destinado a almacenar los frames que la cámara 2D del robot vaya tomando. Simultáneamente, en otro hilo diferente, el detector CNN+MMOD procesa dichos frames y detecta las caras que aparezcan.

El detector está preparado para recibir un conjunto de múltiples frames con varias caras en cada uno de dichos frames. En definitiva, el detector nos devolvería un vector de tres dimensiones siendo la primera el número de frames analizados; la segunda, el número de caras detectadas para cada frame; y, la tercera y última, las coordenadas espaciales de cada esquina de la caja para cada cara.

En nuestro caso, tan solo enviamos al detector un frame por instante de tiempo. A su vez, restringimos la salida del detector a la cara de mayor tamaño, que, en principio, será la del niño cuyo estado de ánimo queramos monitorizar.

3.2 Clasificador de emociones

En nuestra propuesta, empleamos el mismo clasificador que el usado por la aplicación que se encuentra en el repositorio (Enrique Correa, s.f.). A continuación, se justifica el modelo escogido, comentamos su arquitectura, el entrenamiento y el rendimiento de la misma, así como su integración en nuestra aplicación:

3.2.1 Discusión: conjuntos de datos y modelo de la red

Para justificar la elección del modelo que vamos a usar, nos basamos en el artículo que se encuentra en el repositorio, donde se entrenan y comparan tres modelos distintos de clasificador. Para ello, los autores del estudio contemplan varios conjuntos de datos para entrenar los clasificadores:

- Facial Expression Recognition Challenge (FERC-2013).
- Extended Cohn-Kanade (CK+).
- Radboud Faces Database (RaFD).

Varios aspectos diferencian dichos conjuntos de entrenamiento: la calidad , la cantidad y la limpieza de las imágenes. El set FERC-2013, por ejemplo, cuenta con 32000 imágenes de baja resolución, mientras que RaFD contiene 8000 de alta calidad. En cuanto a la limpieza de los sets de entrenamiento, las personas que aparecen en CK+ y RaFD lo hacen posando, mientras

que las imágenes del set FERC-2013 aparecen, digamos, en estado salvaje. Este último aspecto hace que las imágenes de FERC-2013 sean más difíciles de interpretar, pero, considerando el gran tamaño de dicho conjunto de entrenamiento, la diversidad puede mejorar la robustez del modelo a entrenar. Lo anterior se basa en que, una vez entrenado la red mediante FERC-2013, las imágenes de las otras dos datasets, consideradas limpias, pueden ser clasificadas con éxito, mientras que a la inversa, no.

Por todo ello, el set FERC-2013 será el elegido por los autores del artículo para el entrenamiento de la red y su validación; mientras que para el testeo, se hace uso de RaFD para obtener así un indicador de rendimiento en función de datos limpios y de alta calidad.

Los tres modelos que los autores proponen y entrenan son los siguientes:

- La primera red está basada en el trabajo de investigación llevado a cabo previamente en (Krizhevsky, 2012). Dicha red es la más pequeña de las tres, por lo que es también la que menos recursos consume. Debido que nuestra aplicación pretende reconocer emociones en tiempo real, este es un aspecto muy a tener en cuenta.

- En segundo lugar, se prueba con la CNN AlexNet. No obstante, se reduce el tamaño de la red, teniendo en cuenta que el modelo solo debe distinguir entre 7 emociones y buscando una mayor optimización de los recursos. Por tanto, en lugar de cinco, se usan tres capas convolucionales, además de reducirse el número de neuronas de 4096 a 1024 en las tres capas de transformación afín posteriores.

- Por último, se implementa una red basada en el trabajo propuesto en (Gudi, 2015). El único cambio que realizan los autores es el de añadir una capa extra de pooling, con la que reducen notablemente el cómputo sin mermar en exceso la eficacia de la red.

Los tres modelos son entrenados durante 60 épocas mediante 9000 imágenes del set FERC-2013 y validados con otras 1000. Para el testeo, se emplean 1000 imágenes del set de datos RaFD.

Los resultados obtenidos por cada clasificador y las conclusiones que se extraen de estos se muestran a continuación (véase la figura 3.15):

- Para el primer clasificador, el A, los autores obtienen una tasa de validación del 63%. También se observa un aprendizaje considerablemente rápido. Nótese que la capa extra de pooling, aunque reduce bastante el coste computacional, no ocurre lo mismo con la tasa de validación.

- En cuanto al clasificador B, aunque también aprende relativamente rápido, apenas alcanza el 54% de validación, a pesar de ser la red que más recursos consume y de ser, por tanto, la que tiene un comportamiento más lento. Aparentemente, el haber reducido el tamaño de la red penaliza su precisión más de lo esperado.

- Finalmente, el clasificador C muestra un entrenamiento algo más lento, pero una tasa de validación bastante similar a la del modelo A.

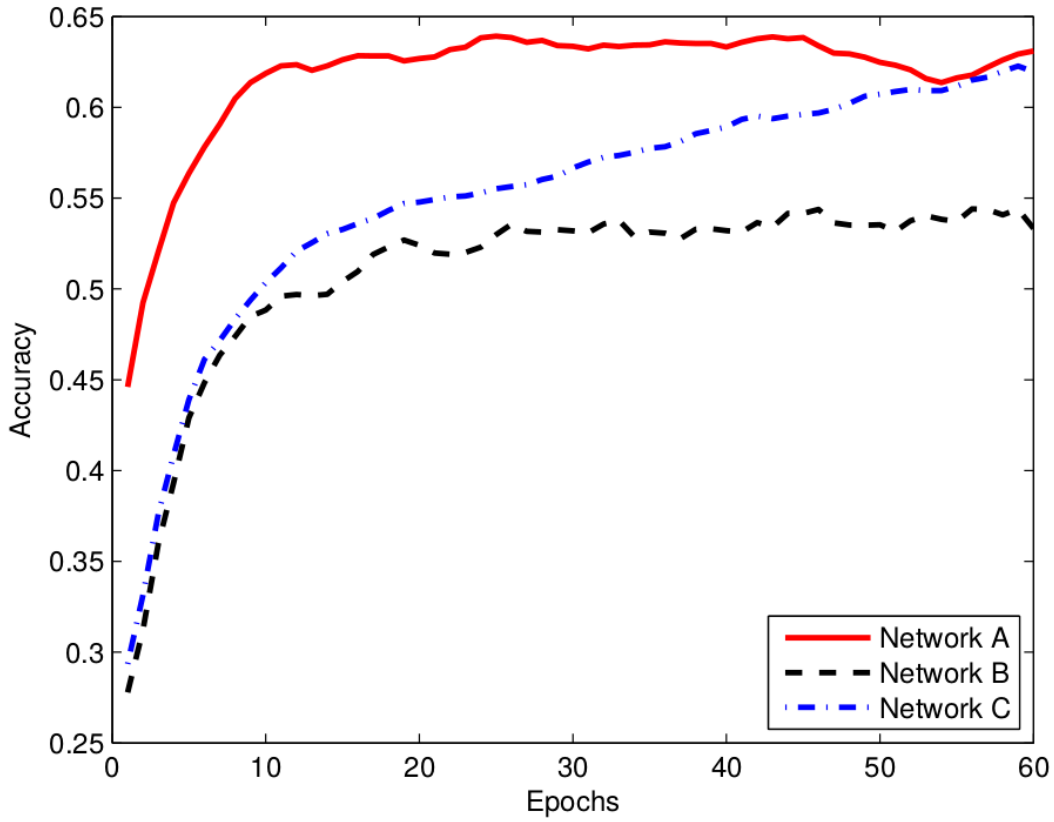


Figura 3.15: Tasa de validación de los distintos clasificadores.

A priori, podría parecer que el clasificador A es el más apropiado. No obstante, al testear los modelos con el set de datos RaFD, la tasa de acierto del clasificador C (60%) es muy superior a la del clasificador A (50%). Por tanto, se deduce que el clasificador A tiene una peor capacidad de generalización que el C, por lo que este último es el que los autores acaban escogiendo para su aplicación.

Si volvemos a fijarnos en la figura 3.15, la precisión del clasificador escogido parece seguir subiendo en sus últimas épocas. Por lo tanto, los autores deciden entrenar el clasificador durante 100 épocas usando, en vez de 9000, 20000 imágenes del set FERC-2013. Con todo, se alcanza una tasa de acierto del **67%**.

Debido a que nuestra aplicación también requiere de una ejecución en tiempo real y de una cierta precisión, nos decantamos también por dicho clasificador.

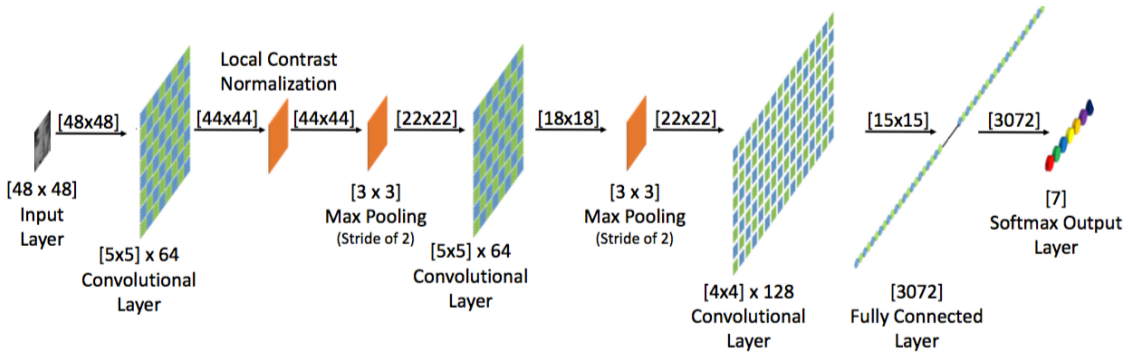


Figura 3.16: Esquema general del clasificador de emociones.

3.2.2 Arquitectura de la red

En la figura 3.16 se muestra un esquema general del modelo final. Comentando en primer lugar la fase de extracción de características, se puede observar que contamos con hasta tres capas convolucionales seguidas de funciones de activación ReLu para obtener características no-lineales de la cara que debemos clasificar. Entre dichas capas, encontramos capas de pooling para reducir la dimensionalidad de la imagen, reduciendo la cantidad de cálculos necesarios y confiriéndole a la red capacidad de abstración.

Al igual que para el detector, de caras de Dlib, observamos que se emplean varias redes convolucionales de tamaño reducido; de 3x3, 4x4 y 5x5 celdas. De nuevo, hacemos esto para obtener características más complejas y expresivas, consumiendo además menos recursos.

En cuanto a la fase de clasificación, se recoge el mapa de características y se procesa mediante una capa de transformación afín conectada a una función Softmax. Dicha función de activación no lineal permite a la red devolver el porcentaje de probabilidad de que una determinada cara pertenezca a las siguientes clases: *alegría*, *sorpresa*, *neutro*, *enfado*, *asco*, *miedo* y *tristeza*. En cambio, usando la función de activación ReLu, solo podríamos detectar si dicha cara aparece o no en la imagen, sin determinar su clase. Sí podríamos implementar un detector por cada emoción para así detectar entre *alegría*/*no-alegría*, neutral/*no-neutral*, etc.; pero lógicamente, esta no sería una solución adecuada.

3.2.3 Eficacia del clasificador

En la gráfica de la figura 3.17, proporcionada por los creadores de la red, se ve el rendimiento de la red por cada emoción. Dicha tabla muestra una muy alta tasa de acierto para *alegría*, *neutro* y *sorpresa*, las cuales son también las más fáciles de reconocer por las personas. En cambio, las emociones *tristeza* y *miedo*, que parecen ser más complicadas de distinguir, son las que peor precisión devuelven.

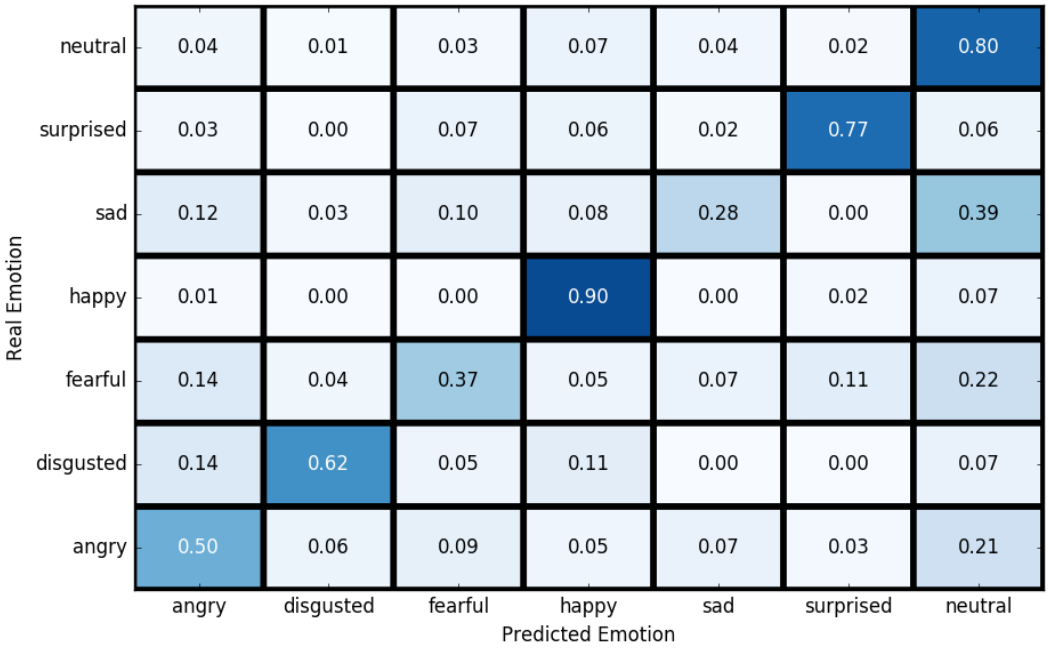


Figura 3.17: Matriz de confusión proporcionada por los creadores de la red.

(a) Cara clasificada como *triste*.

(b) Cara clasificada como *cabreada*.

Figura 3.18: Clasificaciones de ejemplo proporcionadas por los autores.

Para ilustrar esto último, fijémonos en las figuras 3.18a y 3.18b. Para la primera el detector devuelve *triste*, mientras que para la segunda, *cabreado*. No obstante, parece bastante difícil intuir la emoción del sujeto a partir de las fotos. Necesitaríamos un contexto, más información. Podría ayudarnos, por ejemplo, conocer ante qué está reaccionando la persona, su tono de

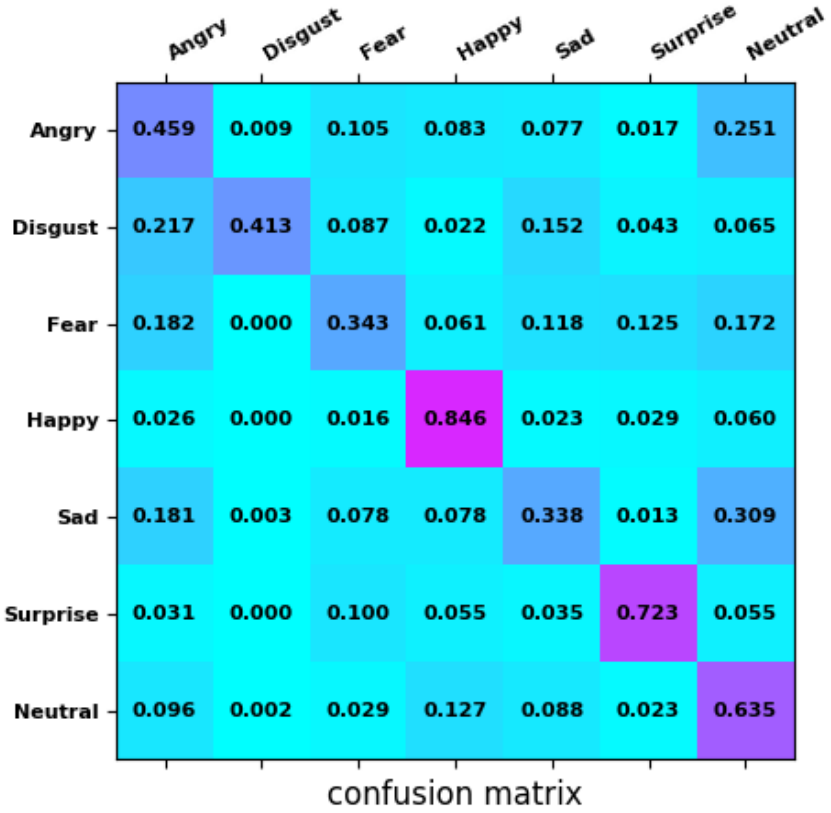


Figura 3.19: Matriz de confusión que obtenemos en el laboratorio.

voz en ese instante, etcétera. Es más, una persona en un determinado instante no suele experimentar una sola emoción, sino una combinación. Podemos sentirnos tristes y enfadados a la vez, incluso contentos pero tristes simultáneamente.

Retomando la cuestión del acierto de la red, aunque los autores reportan un 67%, en las pruebas realizadas en el laboratorio no conseguimos replicar dicho porcentaje. Por el contrario, el mejor resultado que obtenemos es un **53.67%**. En la figura 3.19 podemos observar la matriz de confusión que obtenemos, en función de la cual se calcula el acierto.

Si nos fijamos, las emociones negativas son, de nuevo, las que peores resultados devuelven. Véase el alto porcentaje de confusión entre *tristeza* y *neutro*, por ejemplo.

En definitiva, tal tasa de error es inherente al problema, como ya hemos comentado. En cualquier caso, la red que empleamos obtendría una categoría de *bronce* en el desafío **Kaggle** a fecha de la redacción de este trabajo, quedando en vigésimo puesto aproximadamente (Véase la figura 3.20). Aunque podríamos emplear una red más eficaz, sería a costa de aumentar el coste computacional. De hecho, el clasificador más eficaz de la lista (71% de acierto) emplea una Resnet, que es mucho menos eficiente que la escogida.

19	—	Anil Thomas		0.55363
20	—	dova		0.54444
21	▲1	XterNalz		0.53942
22	▲1	Furstenwald		0.53078
23	▼2	stevenwudi		0.52800

Figura 3.20: Ranking de eficacia del desafío Kaggle.

3.2.4 Integración del clasificador en nuestra aplicación

Dentro de nuestra aplicación, en el mismo hilo creado para el detector, se ejecuta el clasificador de emociones. En caso de detectarse una cara, el clasificador recibe como entrada la subimagen devuelta por el detector con la cara encuadrada y devuelve como salida la emoción cuya probabilidad de acierto sea mayor.

Si nos fijamos en la figura 3.16, las imágenes de entrada a la red deben ser de 48x48 píxeles, por lo que ajustamos el tamaño de la subimagen. Además, la cambiamos a escala de grises antes de enviársela al clasificador.

3.3 Aplicación web

Los lenguajes empleados son HTML (que nos permite mostrar texto e imágenes), CSS (con lo que modificamos la fuente de la letra, el color del fondo, etc.) y JavaScript (que nos permite definir los botones y demás objetos interactivos que vemos en las figuras 3.21 y 3.22).

La aplicación mostrada por la tablet consiste en un cuento interactivo cuya trama se basa en el árbol de decisión que se ve en la figura 3.23.

Contamos con dos tipos diferentes de transiciones:

- **En función de las emociones.** El robot monitoriza en todo momento las emociones del niño y, para cada escena, cuenta cuántas emociones positivas y negativas se dan. Si la escena actual despierta en el niño más emociones positivas que negativas, se toma un camino, y si no, el otro. Véase la figura 3.21.

- **Elección directa.** Se formula una pregunta y es el niño quien de manera directa elige a través de la pantalla táctil qué camino tomar a continuación, tal y como se muestra en la figura 3.22.

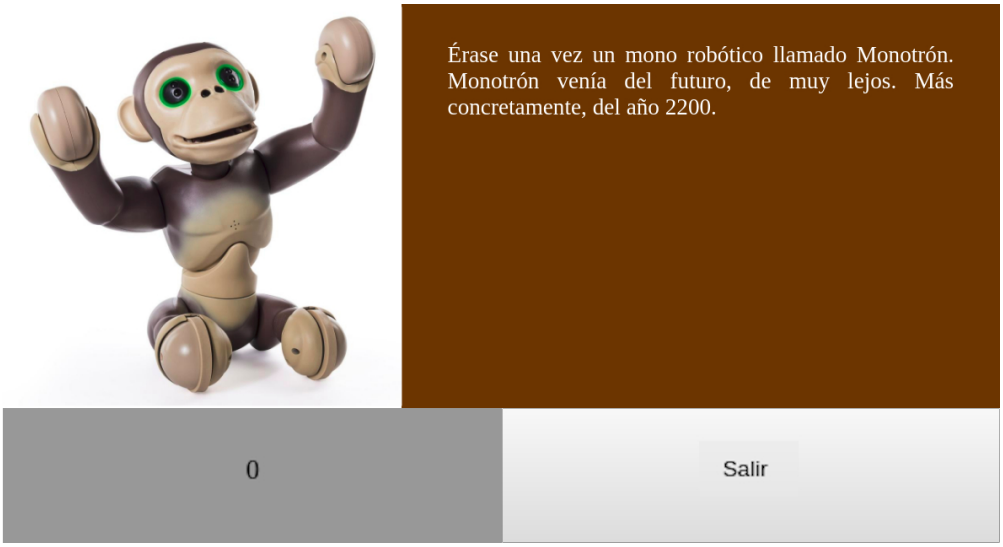


Figura 3.21: Transición en función de las emociones. Escena *index*.

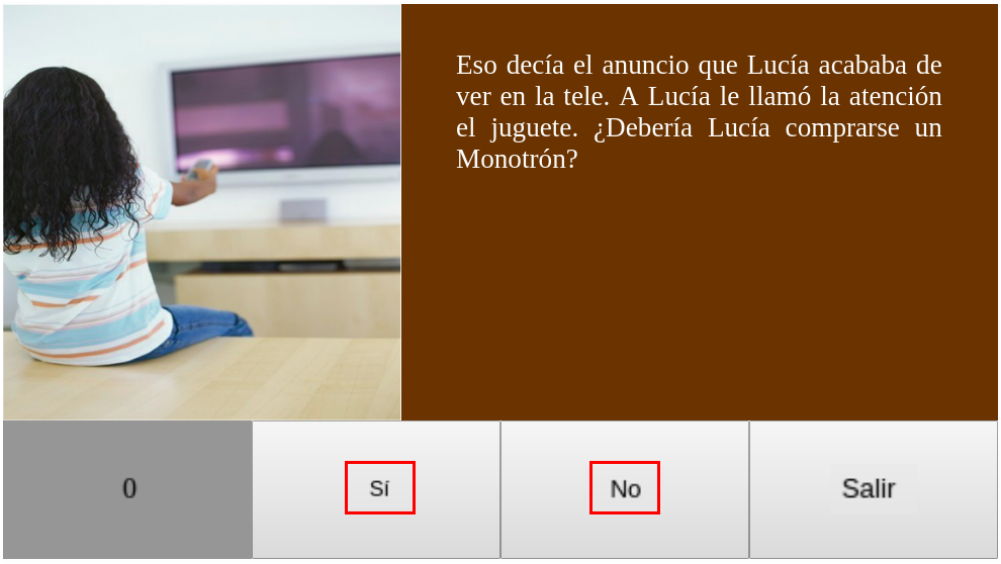


Figura 3.22: Transición en función de elección directa. Escena *indexAB*.

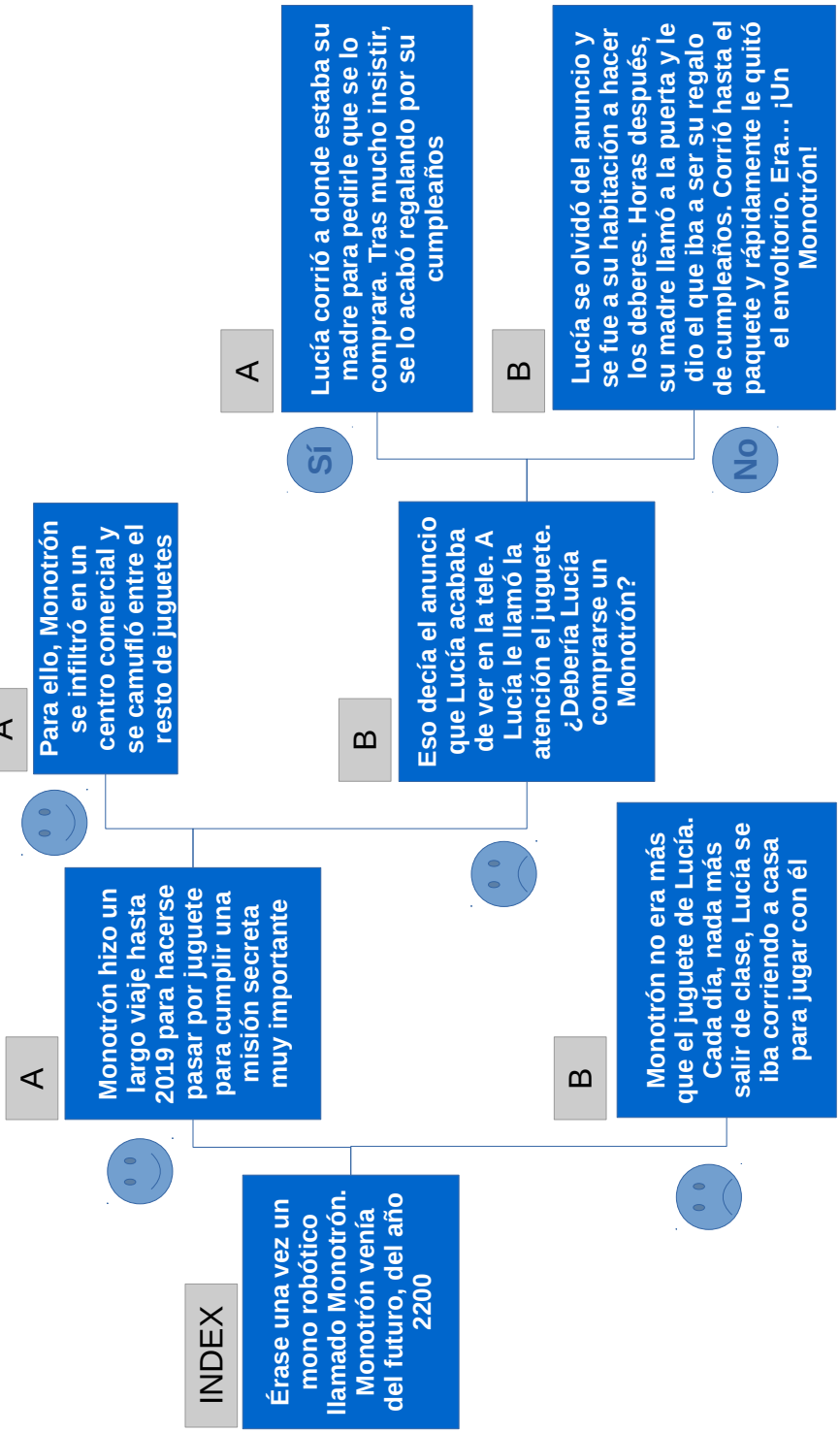


Figura 3.23: Cuento interactivo: Árbol de decisiones.

3.4 Programa principal

Desde el programa principal podremos conectarnos al robot, y controlar sus acciones principales y su comportamiento, todo ello gracias a NAOqi. También será aquí desde donde coordinemos el funcionamiento del resto de la aplicación: cuándo ejecutar una determinada página web en la tablet, qué debe decir en voz alta el robot, etc.

Para ello, en primer lugar, creamos desde nuestro servidor el broker, el ejecutable NAOqi que correrá en el robot. A continuación, nos conectamos a Pepper mediante su dirección IP y, seguidamente, accedemos mediante proxy a los siguientes módulos remotos y a sus respectivos métodos:

- **ALMemory:** Para comprobar que no haya ningún problema con la memoria del robot.
- **ALAutonomousLife**: Sus métodos nos permiten modificar comportamientos básicos del robot como, por ejemplo, si debe o no acompañar con lenguaje corporal lo que dice en voz alta.
- **ALVideoDevice:** El cual nos permite acceder a la cámara del robot y pedirle que tome frames con una frecuencia de 30 fps, con una resolución VGA y a color (RGB).
- **ALFaceDetection:** Con él podemos habilitar el tracking de caras. Es decir, le decimos al robot que siga con la mirada la cara del niño que tenga delante.
- **ALTextToSpeech:** Que nos permitirá hacer que Pepper diga en voz alta el texto deseado.
- **ALTabletService:** Sus métodos nos permiten acceder a la tablet y decidir en todo momento qué página web debe mostrar.

Después, inicializamos dos nuevos hilos de ejecución. El primero deberá acceder al proxy de la cámara y guardar las imágenes que vaya tomando. El segundo hilo estará destinado a tomar dichas imágenes y detectar las caras que aparezcan en ella. En caso de haberlas, se selecciona la de mayor tamaño dentro de la escena y se analiza mediante el clasificador de emociones. Ambos hilos funcionan de manera simultánea sin interrumpir el funcionamiento normal del programa principal.

En la siguiente sección se hablará con mayor profundidad cómo se dirigen desde el programa principal el resto de módulos de la aplicación.

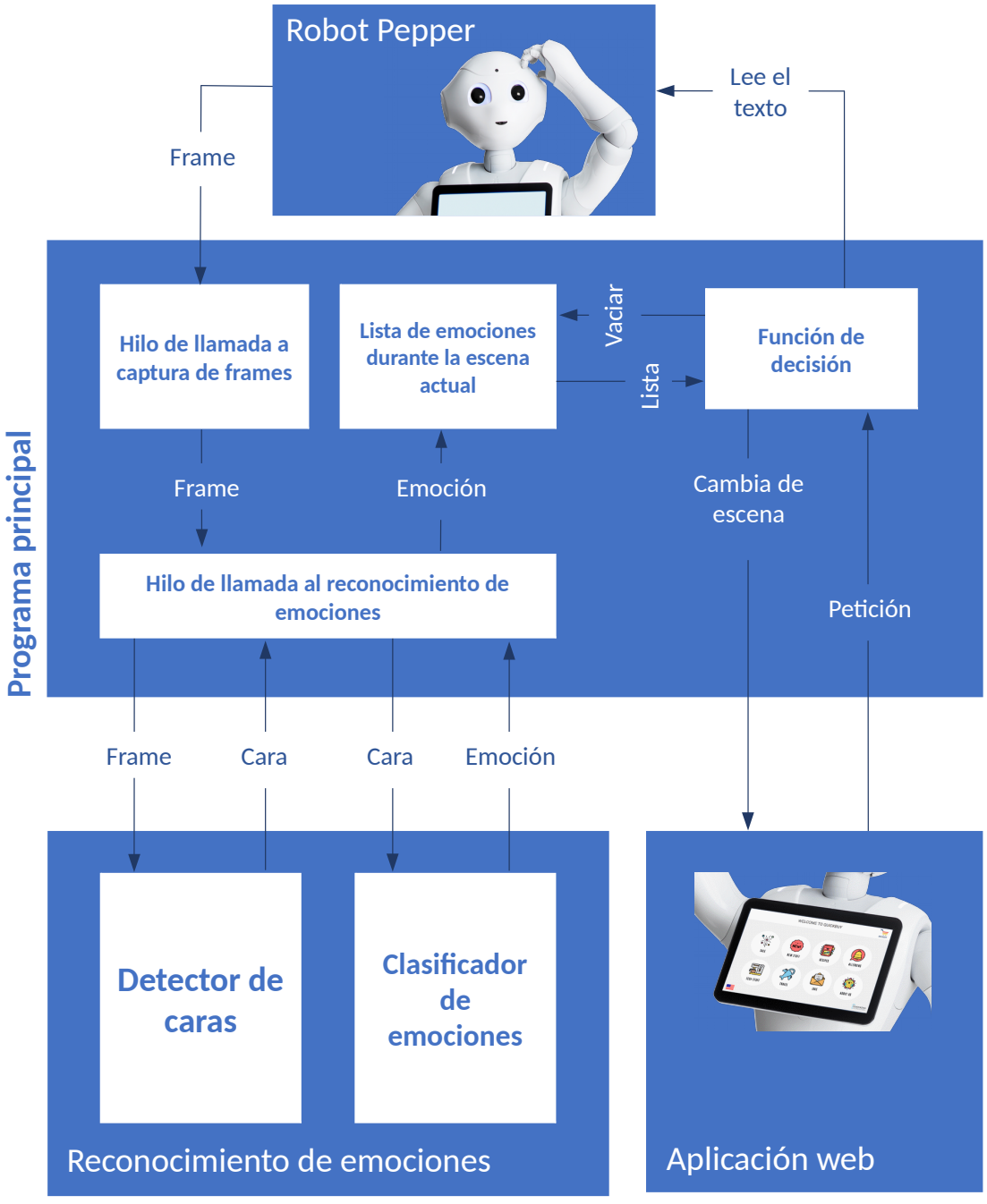


Figura 3.24: Funcionamiento de la aplicación y coordinación de los módulos.

3.5 Coordinación de los módulos

Antes de seguir, cabe mencionar que tanto el programa principal como el módulo de reconocimiento de emociones, se deben ejecutar en una GPU debido al alto consumo de este último.

Veamos a continuación el funcionamiento habitual de la aplicación y cómo se coordinan los distintos módulos. Durante la explicación, véase la figura 3.24:

Encontrándonos en una escena cualquiera, el *hilo de llamada a captura de frames* se encuentra constantemente recibiendo los frames que va tomando la cámara del robot Pepper. Simultáneamente, el *hilo de llamada al reconocimiento de emociones* recibe dichos frames y los envía al *detector de caras*. En caso de que se detecte una cara, se recorta y se guarda en una subimagen, la cual es enviada al *clasificador de emociones*. Desde que empieza la escena actual hasta que acaba, las emociones reconocidas se van añadiendo a la *lista de emociones*.

Paralelamente, el robot muestra por pantalla y lee el fragmento de texto correspondiente con la escena actual. Mientras tanto, la aplicación web puede realizar distintas peticiones, las cuales serán atendidas desde la *función de decisión* del programa principal:

- **Salir de la aplicación:** En cuyo caso, la aplicación web envía la petición **exit:true**, para la cual la *función de decisión* ordena destruir de manera segura todos los objetos y finalizar el programa.

- **Leer el texto:** La petición cuenta con la siguiente estructura: **readText:texto**. La *función de decisión* simplemente ordena al robot que lea el texto en cuestión. Dicha petición se da nada más comenzar una nueva escena, para que el fragmento de la historia sea leído por el robot.

- **Cambiar de escena por elección directa:** Como es el usuario quien elige cuál va ser la escena siguiente, la estructura de la petición debe ser: **finishedPage:escenaActual :escenaSiguiente**. Por ejemplo, una posible petición de este tipo sería: finishedPage:indexAB:A. Ante esta, la *función de decisión* ordena a la aplicación web comenzar la escena siguiente. Además, ordena vaciar la lista de emociones para empezar de cero con la nueva escena.

- **Cambiar de escena en función de las emociones:** La estructura de la petición es **finishedPage:escenaActual**. Por ejemplo: finishedPage:index. Al recibir dicha petición, la *función de decisión* solicita la lista de emociones para la escena actual. Entonces, se clasifican las emociones registradas entre:

  - **Emociones positivas:** Alegría, sorpresa y neutro.

  - **Emociones negativas:** Enfado, asco, miedo y tristeza.

Recordemos que si hay más emociones positivas que negativas, se ordena a la aplicación web cambiar a una escena determinada, tal y como se muestra en la figura 3.23. En caso contrario, se ordena cambiar a la otra escena. Finalmente, igual que en el caso anterior, se ordena vaciar la lista de emociones antes de seguir monitorizando las emociones del niño para la escena siguiente.

Recapitulando, el robot irá leyendo las distintas escenas del cuento y, mientras tanto, irá monitorizando el estado de ánimo del niño y cómo reacciona ante los distintos estímulos que se le presentan. Una vez acabada la escena actual, se elige la que será la siguiente, o bien en función de las emociones del niño, o bien a través de su elección directa.

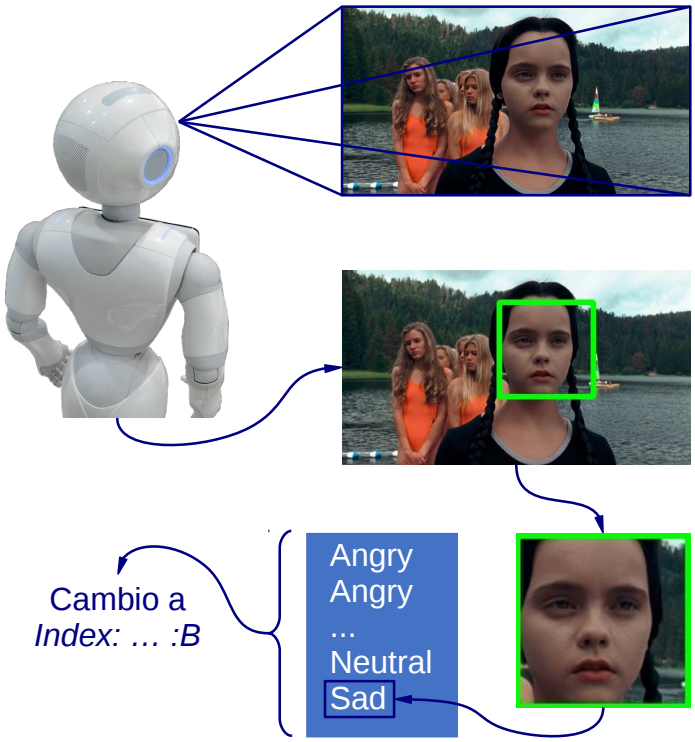


Figura 3.25: Comportamiento habitual del robot.

Por último, en la figura 3.25 se puede observar de una manera más intuitiva el comportamiento más habitual del robot. Es decir, el cambio de escena en función de las emociones:

En primer lugar, vemos que el robot captura un frame, sobre el cual localiza la cara más grande, que, en este caso, es la de Miércoles Addams. A continuación, se toma la cara recortada, clasificada como triste por nuestra aplicación, y se guarda su emoción en la lista correspondiente con la escena actual. Suponiendo que dicho frame es el último captado de la escena, se cuenta el número de emociones de la lista. Como Miércoles no ha disfrutado en exceso del relato, se cuentan más emociones negativas que positivas, por lo que Pepper decide cambiar a la escena *Index: ... :B*.

3.6 Repositorio del proyecto

Tanto el código de la aplicación como un vídeo demostrativo, entre otros, pueden encontrarse en el siguiente repositorio:

https://github.com/guillegallud/tfg-robotica.git

4 Conclusiones

En este trabajo se ha hablado, en primer lugar, de los problemas a los que se enfrentan los niños con autismo a la hora de relacionarse y jugar con sus compañeros, aspectos vitales en el desarrollo personal de cualquier niño.

Debido a que los niños con autismo muestran una mejor interacción con objetos inanimados, se han expuesto distintos trabajos que proponen la incorporación de robots sociales en las terapias con estos niños.

Seguidamente, hemos introducido nuestra propuesta: el desarrollo de una aplicación que permita al robot Pepper contar cuentos sociales interactivos cuya trama evolucione en función de cómo reaccione el niño en cuestión.

Para reconocer las emociones del niño en todo momento y actuar en consecuencia, se ha desarrollado un módulo formado por el detector de caras CNN+MMOD de Dlib y un clasificador de emociones, ambos basados en redes neuronales convolucionales. Además, se ha creado una aplicación web, que se ejecutará en la tablet del robot y mostrará el texto de la historia, además de imágenes que refuercen el relato. Por último, se ha implementado un programa desde el cual comunicarnos mediante NAOqi con el robot y controlar sus acciones, además de coordinar el comportamiento general de la aplicación.

Por último, parte de este trabajo se está considerando para publicación en el **congreso internacional ROBOT2019**: David Azuar, Guillermo Gallud, Felix Escalona, Francisco Gomez-Donoso and Miguel Cazorla. "A Story-telling Social Robot with Emotion Recognition Capabilities for the Intellectually Challenged". ROBOT'2019: Fourth Iberian Robotics Conference. Porto, Portugal. November 2019.

5 Futuras Líneas de Trabajo

En cuanto a futuras mejoras, se nos ocurre lo siguiente:

- **Mejorar la eficiencia del reconocimiento de emociones:** En vez de dividir el reconocimiento de emociones en dos redes distintas (detección de caras y clasificación de emociones), podría implementarse un solo detector de emociones que directamente reconociese caras felices, tristes, etc. a partir de los frames tomados por el robot.

- **Mejorar la eficacia del reconocimiento de emociones:** Es evidente que el análisis de expresiones faciales aisladas resulta insuficiente a la hora de determinar las emociones de una persona. Sería necesario, por tanto, tomar en consideración muchos otros datos, como pueden ser el tono de voz, la evolución de la expresión facial a lo largo de varios frames, el lenguaje corporal, etcétera.

- **Implementar el reconocimiento de voz** para que las transiciones en función de elección directa pudiera producirse también a través de órdenes de voz, lo cual permitiría una mejor adaptación a niños con distintas capacidades. Sería apropiado, por ejemplo, para niños ciegos o que no supieran leer.

- **Mejorar la expresividad del robot:** Podríamos cambiar de color los leds del robot para que acompañasen las sensaciones que Pepper quisiera transmitir. Por ejemplo, si se estuviera contando una historia de terror, los leds cambiarían de color a rojo. También podríamos programar los actuadores del robot para que el mensaje se viera reforzado mediante lenguaje corporal.

Los siguientes pasos a seguir, una vez tratados los aspectos más técnicos de la aplicación, consistirían en **desarrollar y mejorar el cuento social** que hemos propuesto. Para ello, se hace imprescindible la **ayuda de terapeutas** y especialistas en el campo del autismo y trastornos similares. También resultaría de vital importancia la **experimentación del robot con niños con autismo** y contar con el feedback por parte de los profesores de educación especial, si de verdad se pretende que la aplicación sea de utilidad real. En definitiva, lo que está claro es que queda mucho por hacer.

Bibliografía

Aldebaran. (s.f.-a). *Naoqi developer guide: Naoqi framework: Key concepts.* http://doc.aldebaran.com/2-1/dev/naoqi/index.html. (Consultado el 3/7/2019)

Aldebaran. (s.f.-b). *Pepper - developer guide: Technical overview.* http://doc.aldebaran.com/2-4/family/pepper_technical/index_pep.html. (Consultado el 3/7/2019)

APD, R. (s.f.). *¿cuáles son los tipos de algoritmos del machine learning?* https://www.apd.es/algoritmos-del-machine-learning/. (Consultado el 3/7/2019)

Barry, L., y B. Burlew, S. (2004, 02). Using social stories to teach choice and play skills to children with autism. *Focus on Autism and Other Developmental Disabilities - FOCUS AUTISM DEV DISABIL*, *19*, 45-51. doi: 10.1177/10883576040190010601

Bilyea, A., Seth, N., Nesathurai, S., y Abdullah, H. (2017, 07). Robotic assistants in personal care: A scoping review. *Medical Engineering & Physics*, *49*. doi: 10.1016/j.medengphy .2017.06.038

Briega, R. L. (2017, apr). *Introducción a la inteligencia artificial.* https://relopezbriega.github.io/blog/2017/06/13/introduccion-al-deep-learning/. (Consultado el 3/7/2019)

Brownlee, J. (s.f.). *A gentle introduction to pooling layers for convolutional neural networks.* https://machinelearningmastery.com/pooling-layers-for-convolutional-neural-networks/. (Consultado el 3/7/2019)

Caparrini, F. S. (s.f.). *Introducción al aprendizaje automático.* http://www.cs.us.es/ fsancho/?e=75. (Consultado el 3/7/2019)

Davis, C. (s.f.). *When is a large-sized kernel useful in cnn?* https://www.quora.com/When-is-a-large-sized-kernel-useful-in-CNN. (Consultado el 3/7/2019)

Dertat, A. (s.f.). *Applied deep learning.* https://towardsdatascience.com/applied-deep-learning-part-1-artificial-neural-networks-d7834f67a4f6. (Consultado el 3/7/2019)

Dlib. (s.f.). *Dlib c++ library.* http://dlib.net/. (Consultado el 3/7/2019)

Dwiyantoro, A. P. J. (s.f.-a). *The evolution of computer vision techniques on face detection.* https://medium.com/nodeflux/the-evolution-of-computer-vision-techniques-on-face-detection-part-2-4af3b22df7c2. (Consultado el 3/7/2019)

Dwiyantoro, A. P. J. (s.f.-b). *Performance showdown of publicly available face detection model.* https://medium.com/nodeflux/performance-showdown-of-publicly-available-face-detection-model-7c725747094a. (Consultado el 3/7/2019)

Enrique Correa, M. O. R. S., Arnoud Jonker. (s.f.). *Emotion recognition using dnn with tensorflow.* https://github.com/isseu/emotion-recognition-neural-networks. (Consultado el 3/7/2019)

Gildenblat, J. (s.f.). *Hallucinating faces with dlib's face detector model in pytorch.* https://jacobgil.github.io/deeplearning/hallucinating_faces_dlib_pytorch. (Consultado el 3/7/2019)

Gudi, A. (2015). Recognizing semantic features in faces using deep learning. *ArXiv*, *abs/1512.00743*.

Gupta, V. (s.f.). *Face detection - opencv, dlib and deep learning (c++ / python).* https://www.learnopencv.com/face-detection-opencv-dlib-and-deep-learning-c-python/. (Consultado el 3/7/2019)

Hulstaert, L. (s.f.). *https://towardsdatascience.com/going-deep-into-object-detection-bed442d92b34.* https://towardsdatascience.com/going-deep-into-object-detection-bed442d92b34. (Consultado el 3/7/2019)

Js-overview. (s.f.). https://www.afterhoursprogramming.com/tutorial/javascript/javascript-overview/. (Consultado el 3/7/2019)

Karpathy, A. (s.f.). *Cs231n convolutional neural networks for visual recognition: Convnet architectures: Layer patterns.* http://cs231n.github.io/convolutional-networks/#layerpat. (Consultado el 3/7/2019)

King, D. (s.f.-a). *dlib-models.* https://github.com/davisking/dlib-models. (Consultado el 3/7/2019)

King, D. (s.f.-b). *Dnn mmod example.* http://dlib.net/dnn_mmod_ex.cpp.html.

King, D. (s.f.-c). *Dnn mmod for cars example.* http://dlib.net/dnn_mmod_train_find_cars_ex.cpp.html. (Consultado el 3/7/2019)

King, D. (s.f.-d). *Dnn mmod for faces example.* http://dlib.net/dnn_mmod_face_detection_ex.cpp.html. (Consultado el 3/7/2019)

King, D. (s.f.-e). *Easily create high quality object detectors with deep learning.* http://blog.dlib.net/2016/10/easily-create-high-quality-object.html.

King, D. (s.f.-f). *Vehicle detection with dlib 19.5.* http://blog.dlib.net/2017/08/vehicle-detection-with-dlib-195_27.html. (Consultado el 3/7/2019)

Krizhevsky, A. (2012, 05). Learning multiple layers of features from tiny images. *University of Toronto.*

Kumar, A. (s.f.). *Machine learning - 7 steps to train a neural network.* https://vitalflux.com/machine-learning-7-steps-train-neural-network/. (Consultado el 3/7/2019)

Murphy, J. (2016). An overview of convolutional neural network architectures for deep learning..

NVIDIA. (s.f.-a). *Artificial neural network.* https://developer.nvidia.com/discover/artificial-neural-network#neural-network-inference. (Consultado el 3/7/2019)

NVIDIA. (s.f.-b). *Conv neural network.* https://developer.nvidia.com/discover/convolutional-neural-network. (Consultado el 3/7/2019)

Robins, B., Dautenhahn, K., y Dubowski, J. (2006, 01). Does appearance matter in the interaction of children with autism with a humanoid robot? interact stud. *Interaction Studies*, *7*. doi: 10.1075/is.7.3.16rob

Robins, B., Dickerson, P., Hyams, P., y Dautenhahn, K. (2004, 01). Robot-mediated joint attention in children with autism: A case study in robot-human interaction. *Interaction Studies*, *5*, 161-198. doi: 10.1075/is.5.2.02rob

Saha, S. (s.f.). *A comprehensive guide to convolutional neural networks - the eli5 way.* https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53. (Consultado el 3/7/2019)

Sanoja, C. (s.f.). *Actuadores.* https://github.com/CarSanoja/KyoFridge_EC3882/wiki/6.-Actuadores. (Consultado el 3/7/2019)

Schmidhuber, J. (2015). Deep learning in neural networks: An overview. *Neural Networks*, *61*, 85 - 117. Descargado de `http://www.sciencedirect.com/science/article/pii/S0893608014002135` doi: https://doi.org/10.1016/j.neunet.2014.09.003

Shah Johan Hamzah, M., Shamsuddin, S., Azfar Miskam, M., Yussof, H., y Said Hashim, K. (2014, 12). Development of interaction scenarios based on pre-school curriculum in robotic intervention for children with autism. *Procedia Computer Science*, *42*, 214 - 221. doi: 10.1016/j.procs.2014.11.054

Shamsuddin, S., Yussof, H., Ismail, L., Hanapiah, F., Mohamed, S., Ali Piah, H., y Ismarrubie Zahari, N. (2012, 03). Initial response of autistic children in human-robot interaction therapy with humanoid robot nao. *Proceedings - 2012 IEEE 8th International Colloquium on Signal Processing and Its Applications, CSPA 2012*. doi: 10.1109/CSPA.2012.6194716

Trejo, J. (2017). *Mercadotecnia digital: Una descripción de las herramientas que apoyan la planeación estratégica de toda innovación de campaña web.* Grupo Editorial Patria. Descargado de `https://books.google.es/books?id=AUbJDgAAQBAJ`

Vanderborght, B., Simut, R., Saldien, J., Pop, C., Rusu, A., Pintea, S., … O. David, D. (2012, 01). Using the social robot probo as a social story telling agent for children with asd. *Interaction Studies*, *13*. doi: 10.1075/is.13.3.02van

Villa, D. (2008, 4). *Latex: Listados de código cómodos y resultones con listings.* http://crysol.org/es/node/909. Descargado 12/12/2014, de `http://crysol.org/es/node/909`

Virtual, I. (s.f.). *Conceptos básicos sobre tecnologías de desarrollo web.* https://www.ingeniovirtual.com/conceptos-basicos-sobre-tecnologias-de-desarrollo-web/. (Consultado el 3/7/2019)

Wada, K., Shibata, T., Musha, T., y Kimura, S. (2008, 08). Robot therapy for elders affected by dementia. *Engineering in Medicine and Biology Magazine, IEEE*, *27*, 53 - 60. doi: 10.1109/MEMB.2008.919496

What's javascript? (s.f.). http://www.tutorialspoint.com/javascript/javascript_overview.htm. (Consultado el 3/7/2019)

Wolniansky, P. W., Foschini, G. J., Golden, G., y Valenzuela, R. A. (1998). V-blast: An architecture for realizing very high data rates over the rich-scattering wireless channel. En *Signals, systems, and electronics, 1998. issse 98. 1998 ursi international symposium on* (pp. 295–300).